算法之美

隐匿在数据结构背后的原理

左飞 / 著

C++版

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

VISIT…

内 容 简 介

本书围绕算法与数据结构这个话题，循序渐进、深入浅出地介绍了现代计算机技术中常用的 40 余个经典算法，以及回溯法、分治法、贪婪法和动态规划等算法设计思想。在此过程中，本书也系统地讲解了链表（包括单向链表、单向循环链表和双向循环链表）、栈、队列（包括普通队列和优先级队列）、树（包括二叉树、哈夫曼树、堆、红黑树、AVL 树和字典树）、图、集合（包括不相交集）与字典等常用数据结构。同时，通过对 22 个经典问题（包括约瑟夫环问题、汉诺塔问题、八皇后问题和骑士周游问题等）的讲解，逐步揭开隐匿在数据结构背后的算法原理，力图帮助读者夯实知识储备，激活思维技巧，并最终冲破阻碍编程能力提升的重重藩篱。

本书适合作为大专院校相关专业学生研习算法与数据结构知识的课外参考书。对有意参加信息学竞赛的读者，本书亦有很强的参考价值。此外，鉴于算法与数据结构在求职过程中常常被视为考察重点，所以就临近毕业的学生或其他欲从事 IT 行业的求职者而言，阅读本书也将对面试备考大有裨益。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

算法之美：隐匿在数据结构背后的原理：C++版 / 左飞著. —北京：电子工业出版社，2016.3
ISBN 978-7-121-27718-4

Ⅰ. ①算… Ⅱ. ①左… Ⅲ. ①数据结构－程序设计②C 语言－程序设计 Ⅳ. ①TP311.12②TP312

中国版本图书馆 CIP 数据核字(2015)第 286959 号

责任编辑：付　睿
印　　刷：
装　　订：
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　邮编：100036
开　　本：787×1092　1/16　印张：26.75　字数：617.6 千字
版　　次：2016 年 3 月第 1 版
印　　次：2016 年 3 月第 1 次印刷
印　　数：3000 册　　定价：79.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。

前言

2014 年的冬天，一部讲述计算机科学之父艾伦·图灵传奇人生的传记电影在美国上映，这部影片就是《模仿游戏》。次年，该片荣获第 87 届奥斯卡金像奖最佳改编剧本奖，以及包括最佳影片、最佳导演、最佳男主角、最佳女配角在内的 7 项提名，一时风光无限。尽管现代计算机已经无处不在，但因图灵的时代离我们过于久远，现今人们对他的研究工作已经知之甚少。

要说起图灵的贡献，我们还得把时间再往前推。1900 年，德国数学家大卫·希尔伯特在巴黎举行的国际数学家大会上做了题为《数学问题》的演讲，在这篇重要的演讲中，他提出了著名的希尔伯特之 23 个问题。尽管此后的数学发展远远超过了希尔伯特的预料，但他所提出的 23 个问题仍然对 20 世纪的数学发展起到了非常积极的推动作用。

希尔伯特的第 10 个问题是要设计一个算法来测试多项式是否有整数根。他没有使用算法这个术语，而是采用了下面这种表述："通过有限多次运算就可以决定的过程"。有意思的是，从希尔伯特对这个问题的陈述可以看出，他明确地要求设计一个算法。因此，他显然是假设这样的算法是存在的，人们所要做的只是找到它。现在我们知道，这个任务是无法完成的，即它是算法上不可解的。但对那个时期的数学家来说，以他们对算法的直观认识，得出这样的结论是不可能的。

非形式地说，算法是为实现某个任务而构造的简单指令集。以日常用语来说，算法又称为过程或者方法。算法在数学中也起着非常重要的作用。古代数学文献中就包含有执行各种各样计算任务的算法描述。例如，我国古代数学经典《九章算术》中就记述了包括求最大公约数、最小公倍数、开平方根、开立方根等在内的诸多算法。现代计算机科学中更是充满了各种各样的算法。例如，求解最短路径的狄克斯特拉算法，进行字符串匹配的 KMP 算法等。

虽然算法在数学中已有很长的历史，但在 20 世纪之前，算法概念本身一直没有精确的定义。数学家们面对希尔伯特的第 10 个问题，显得束手无策。由于缺乏对于算法本身的精确定义，所以要证明某个特定任务不存在算法则完全不可能。要想破解希尔伯特的第 10 个问题，人们不得不等待算法之精确定义的出现。

直到 1936 年，曙光似乎出现了。图灵向伦敦的权威数学杂志递交了一篇题为《论数字计算在决断难题中之应用》的论文。该文最终于 1937 年正式发表，并立即引起了广泛的注意。在论文中，图灵描述了一种可以辅助数学研究的机器，也就是后来被称为"图

灵机”的抽象系统。与此同时，另外一位数学家阿隆佐·丘奇也独立地提出了另外一套系统，即所谓的λ演算。图灵采用他的图灵机来定义算法，而丘奇则采用λ演算来定义算法，后来图灵证明这两个定义是等价的。由此，人们在算法的非形式概念和精确定义之间建立了联系，即算法的直觉概念等价于图灵机算法，这就是所谓的丘奇－图灵论题。

丘奇－图灵论题提出的算法定义是解决希尔伯特第 10 个问题所必需的。而第 10 个问题的真正解决则要等到 1970 年，借助于丘奇与图灵的杰出贡献，马提亚塞维齐在戴维斯、普特纳姆和罗宾逊等人工作的基础上，最终证明检查多项式是否有整数根的算法是不存在的。

从图灵开始，算法已然同计算机科学之间产生了密不可分的联系。当然，本书的内容并不打算从图灵机开始讲起。回顾建立算法形式化定义和破解希尔伯特第 10 个问题的那段风起云涌的历史，更多地是想说明算法之于我们的世界是多么重要。

无论你是信息技术的从业人员，还是计算机专业的在校学生，再或者是从事相关专业的研究人员，熟练掌握一门计算机语言的重要性都不言而喻。但是不是掌握了这其中的语法规则就能写出漂亮的程序了呢？答案当然是否定的。因为你还需要另外一样至少同等重要的工具——算法。算法和语言的关系，其实很像是“道”和“术”的关系。掌握一门语言，就如同习得一门技艺，可以成为一名工匠。但要想从工匠一跃成为大师，单单停留在“术”的层面显然不够，更重要的是悟“道”。而算法无疑就是计算机程序设计中的“道”。

谈到算法的重要性就不得不提及计算机科学家安德鲁·艾派尔在 1985 年所开展的一项研究工作，这也是程序性能优化领域的经典案例。彼时，艾派尔编写了一个用于计算重力场中天体间相互作用之问题的程序。给定场中物体质量、初始位置和速度等条件，该程序即可对 10000 个天体相互作用时其中两个天体的运行状态进行模拟和仿真。由于计算量太大，最初的程序要完成该项计算大约需要耗时一年。在一系列的改进之后，艾派尔最终将程序耗时有效地缩短到了一天！而在这个改进过程中，算法和数据结构的调优占了主要比重。

再说一个发生在笔者身上的例子。曾经在上学的时候，老师布置了一道编程作业，用于模拟一个猜三和弦的游戏。一个三和弦是指从 A、B、C、D、E、F、G 这 7 个音中任选 3 个组成的一个旋律，而每个音又有高音、中音、低音 3 种情况（分别用 1、2、3 来表示）。现在假设一名作曲家心中有了一个心仪的旋律，然后一个钢琴演奏者试图猜测这个答案。每当演奏者给出一个猜测，例如“A1、B2、C3”。那么作曲家将只能答复这其中完全猜中的音调（即音符和音高都猜对）有几个，除了完全猜中的音调以外，音符猜中了几个，音高猜中了几个。然后演奏者继续猜测，直到完全猜中为止。要知道，全部的组合可能有 1330 种之多！而我们希望用越少的次数猜中越好。不知道本书的各位读者心中是否已想到什么方法来解决这个问题。不过笔者最终实现的程序可以做到平均 4.2 次便猜中答案。而在这个过程中，设计一个绝佳的算法无疑是不二之选。

说起算法又不得不提及数据结构，二者是相辅相成、密不可分的。一方面，算法一定要借助相应的数据结构才能得以实现，另一方面我们在定义一个数据结构的同时其实也已经定义了与之相关的操作。这些操作本身执行的步骤就是算法。

总的来说，本书围绕算法与数据结构这个话题，循序渐进、深入浅出地介绍了现代计算机技术中常用的 40 余个经典算法（包括模式匹配算法、排序算法、散列算法、最短路径算法等），以及回溯法、分治法、贪婪法和动态规划等算法设计思想。本书也系统地讲解了链表（包括单向链表、单向循环链表和双向循环链表）、栈、队列（包括普通队列和优先级队列）、树（包括二叉树、哈夫曼树、堆、红黑树、AVL 树和字典树）、图、集合（包括不相交集等）与字典等常用数据结构。同时，通过对 22 个经典问题（包括约瑟夫环问题、汉诺塔问题、八皇后问题和骑士周游问题等）的讲解，逐步揭开隐匿在数据结构背后的算法原理，力图帮助读者夯实知识储备，激活思维技巧，并最终冲破阻碍编程能力提升的重重藩篱。

更值得一提的是，算法与数据结构知识是技术类求职过程中的必考内容。希望广大读者，尤其是处于求职应聘阶段的毕业生，在夯实基础、培养能力的同时，亦能设法将知识转化为生产力，求得一份称心如意的职位。若能事半功倍、一石二鸟，何乐而不为？为此，笔者特别在附录中整理出了一些求职面试中的经典题目，供有相关需求的读者参考学习。该套题目主要以算法与数据结构问题为主线，并穿插以 C/C++相关的编程问题，具有较高的实用性，对提高应聘竞争力很有帮助。特别地，在正文中涉及相关考点之处，笔者均采用旁注的形式点明了可以参考的题目编号，便于读者在阅读过程中，边学边练，知行合一。

纸上得来终觉浅，绝知此事要躬行。锤炼数据结构的运用能力和深化算法思想的理解程度都有赖于编程实践活动。本书采用 C++作为描述语言，并提供有涉及的全部数据结构和算法之实现代码，供读者参考学习。这些代码均在基于 TDM-GCC 4.9.2 的 DEV-C++ 5.11 和 Visual Studio 2013 环境下编译通过。本书特别提供了一个在线支持资源，地址 http://blog.csdn.net/baimafujinji，从中读者可以下载得到全书的配套代码和附录题库的参考答案，本书的勘误也将实时发布在此博客上。同时欢迎读者就本书中的问题和不足与笔者展开讨论，有关问题请在上述博客中留言。

最后，刘航、吴凯、姜萌、何鹏、胡俊、李召恒、初甲林等人也参与了本书编写工作，笔者在此表示由衷的感谢。

自知论道须思量，几度无眠一文章。由于时间和能力有限，书中纰漏在所难免，真诚地希望各位读者和专家不吝批评斧正。

目录

第 1 章　从数据到算法 1

1.1　数据与数据结构 1

1.1.1　数据及其类型 1

1.1.2　数据结构简介 3

1.2　算法 5

1.2.1　算法的概念 5

1.2.2　算法的分析 8

1.2.3　算法的设计 12

1.3　C++中的 STL 18

1.3.1　STL 简介 19

1.3.2　STL 构成 20

1.3.3　STL 的不同版本 22

本章参考文献 23

第 2 章　指针与数组——也谈中国古代兵制 24

2.1　指针 24

2.1.1　内存与地址 24

2.1.2　指针的语法 27

2.1.3　使用指针变量 29

2.1.4　函数与参数传递 31

2.2　数组 36

2.2.1　结构型数据类型 37

2.2.2　数组定义与初始化 37

2.2.3　数组与指针 41

2.2.4　数组的抽象数据类型 45

2.3　数组应用举例 48

2.3.1　Z 字形编排问题 48

2.3.2　大整数乘法问题 51

2.3.3　九宫格问题 52

2.4　动态内存管理 53

2.4.1　关键词 new 和 delete 53

2.4.2　避免内存错误 56

本章参考文献 61

第 3 章　字符串与模式匹配——梦里寻她千百度 62

3.1　基本概念与定义 62

3.1.1 C++中的字符串 62
3.1.2 字符串抽象数据类型 65
3.2 文本的精确匹配 66
3.2.1 BF 算法 66
3.2.2 MP 算法 67
3.2.3 KMP 算法 72
3.2.4 BM 算法 75
3.2.5 BMH 算法 81
3.3 文本的模糊匹配 83
3.3.1 全局编辑距离 83
3.3.2 局部最佳对准 86
3.3.3 *N* 元距离模型 87
3.3.4 语音编码模型 88
本章参考文献 89
第 4 章 链表——老鹰捉小鸡 91
4.1 链表的概念 91
4.2 单向链表 92
4.2.1 单向链表的结构 92
4.2.2 单向链表的操作算法 94
4.2.3 有序链表的合并算法 101
4.3 单向循环链表 102
4.3.1 单向循环链表的结构 102
4.3.2 单向循环链表的实现 103
4.3.3 约瑟夫环的问题 107
4.3.4 魔术师发牌问题 108
4.3.5 拉丁方阵的问题 109
4.4 双向循环链表 110
4.4.1 双向循环链表的结构 110
4.4.2 双向循环链表的实现 111
4.4.3 维吉尼亚加密法问题 115
4.5 游标类的设计与实现 117
4.5.1 游标类的结构 117
4.5.2 游标类的实现 118
4.6 STL 与链表 122
4.6.1 STL 中链表类的接口 122
4.6.2 遍历 124
4.6.3 元素的插入与删除 125
本章参考文献 126
第 5 章 先进先出与后进先出——简单而深刻 127
5.1 摞盘子的策略 127
5.1.1 栈的结构 127
5.1.2 栈的操作及实现 129
5.1.3 括号匹配问题 132
5.1.4 停车场模拟问题 133
5.2 排队的智慧 136

5.2.1 队列的结构 136
5.2.2 队列的操作及实现 138
5.2.3 舞伴问题 142
5.2.4 杨辉三角问题 143
5.2.5 游程编码问题 145
5.3 优先级队列——兼谈页面置换算法 146
5.3.1 优先级队列的结构 146
5.3.2 优先级队列的实现 149
5.4 STL 中的栈与队列 150
5.4.1 STL 中的 stack 151
5.4.2 STL 中的 queue 153
5.4.3 STL 中的 priority_queue 155
本章参考文献 158
第 6 章 递归——老和尚讲故事 159
6.1 递归的概念 159
6.1.1 定义 159
6.1.2 应用递归的原则 162
6.1.3 递归和非递归的转化 168
6.2 分治法 170
6.2.1 分治法简述 171
6.2.2 汉诺塔问题 172
6.2.3 传染病问题 174
6.3 回溯法 176
6.3.1 回溯法简述 176
6.3.2 迷宫问题 176
6.3.3 八皇后问题 180
本章参考文献 183
第 7 章 树——从红楼梦说起 184
7.1 认识树这种结构 184
7.1.1 基本定义 184
7.1.2 一些术语 186
7.1.3 树的抽象 187
7.2 花开二枝分外香——二叉树及相关算法 188
7.2.1 二叉树的定义 188
7.2.2 二叉树的性质 190
7.2.3 二叉树的实现 191
7.2.4 二叉树的遍历算法 196
7.2.5 二叉树线索化算法 200
7.3 合抱之木，生于毫末——从树到森林 203
7.3.1 树的存储表示 203
7.3.2 树的实现 206
7.3.3 树与森林的遍历算法 209
7.3.4 森林与二叉树的转换 211
7.4 哈夫曼树——最优二叉树编码算法 213
7.4.1 哈夫曼编码 213

7.4.2 构造哈夫曼树 215
7.4.3 哈夫曼编码的实现 216
7.5 堆 220
7.5.1 堆的概念 220
7.5.2 堆的建立 221
7.5.3 堆的操作 223
7.6 基于 STL 实现树结构 224
7.6.1 STL 中的 vector 224
7.6.2 STL 中的 map 228
本章参考文献 230
第 8 章 图——始于哥尼斯堡的七桥问题 231
8.1 图的基本概念 231
8.1.1 图的定义 231
8.1.2 图的术语 232
8.1.3 图的运算 236
8.1.4 图的抽象数据类型 237
8.2 图的存储与表示 239
8.2.1 图的邻接矩阵表示 239
8.2.2 图的邻接表表示 241
8.2.3 两种表示法的比较 243
8.3 图的遍历 244
8.3.1 欧拉路径与欧拉回路 244
8.3.2 哈密尔顿路径与哈密尔顿回路 248
8.3.3 广度优先遍历算法 252
8.3.4 深度优先遍历算法 254
8.4 最短路径问题 258
8.4.1 固定起点最短路径问题 258
8.4.2 非固定起点最短路径问题 264
8.4.3 最短路径的动态规划解法 266
8.5 最小生成树 273
8.5.1 最小生成树的定义 273
8.5.2 克鲁斯卡尔算法 275
8.5.3 普里姆算法 279
本章参考文献 283
第 9 章 树形搜索结构——做一名出色的园艺师 284
9.1 二叉搜索树 284
9.1.1 二叉搜索树的概念 284
9.1.2 二叉搜索树的操作 285
9.1.3 二叉搜索树的实现 288
9.1.4 二叉搜索树的分析 291
9.2 自平衡的二叉搜索树——AVL 树 294
9.2.1 AVL 树的概念 294
9.2.2 AVL 树的旋转 295
9.2.3 AVL 树的实现 299
9.3 树中亦有“红与黑” 303

9.3.1 红黑树的概念 303
9.3.2 红黑树的操作 306
9.3.3 红黑树的实现 314
9.4 基于 Trie 树的单词检索 314
9.4.1 Trie 树的概念 315
9.4.2 Trie 树的表示 316
9.4.3 Trie 树的实现 317
本章参考文献 320
第 10 章 集合与字典——再言搜索之话题 321
10.1 集合论基础 321
10.1.1 集合的概念 321
10.1.2 集合的运算 323
10.2 集合的实现 325
10.2.1 位向量集合 325
10.2.2 单链表集合 330
10.3 字典 337
10.3.1 字典的概念 338
10.3.2 搜索运算 342
10.4 散列 346
10.4.1 散列的概念 347
10.4.2 散列函数 348
10.4.3 字符串散列 351
10.4.4 处理散列冲突 353
10.5 拼写检查问题 358
10.6 不相交集 363
10.6.1 不相交集的概念 363
10.6.2 不相交集的实现 366
10.6.3 犯罪团伙的问题 369
10.6.4 路径压缩的实现 370
10.7 STL 中的 set 371
本章参考文献 374
第 11 章 排序——有序让世界更美好 375
11.1 排序问题概述 375
11.1.1 基本概念和定义 375
11.1.2 排序算法的分类 376
11.1.3 排序算法的分析 377
11.2 插入排序 378
11.2.1 直接插入排序 378
11.2.2 二分插入排序 380
11.2.3 希尔排序 382
11.3 选择排序 384
11.3.1 直接选择排序 384
11.3.2 堆排序 386
11.4 交换排序 390

11.4.1 冒泡排序 390
11.4.2 鸡尾酒排序 392
11.4.3 快速排序 395
11.5 归并排序 399
11.6 计数排序 403
本章参考文献 407
附录 经典求职面试题目 408

45 个算法

BF 算法 66
MP 算法 67
KMP 算法 72
BM 算法 75
BMH 算法 81
德勒曼-温施算法 83
史密斯-沃特曼算法 86
N-gram 算法 87
Soundex 算法 88
Phonix 算法 89
折半/二分查找算法 163
欧几里得算法 170
二叉树的遍历算法 196
哈夫曼算法 215
广度优先遍历算法 252
深度优先遍历算法 254
狄克斯特拉算法 259
弗洛伊德算法 264
最短路径的动态规划算法 266
克鲁斯卡尔算法 275
普里姆算法 279
AVL 树的旋转算法 295
红黑树的操作算法 306
直接定址法 348
除留余数法 348
平方取中法 349
乘余取整法 350
折叠法 350
BKDR 散列算法 351
RS 散列算法 351

FNV 散列算法 351
线性探查法 354
二次探查法 355
双重散列法 356
并查集的路径压缩算法 370
直接插入排序算法 378
二分插入排序算法 380
希尔排序算法 382
直接选择排序算法 384
堆排序算法 386
冒泡排序算法 390
鸡尾酒排序算法 392
快速排序算法 395
归并排序算法 399
计数排序算法 403

22 个经典问题

Z 字形编排问题 48
大整数乘法问题 51
九宫格问题 52
约瑟夫环的问题 107
魔术师发牌问题 108
拉丁方阵的问题 109
维吉尼亚加密法问题 115
括号匹配问题 132
停车场模拟问题 133
舞伴问题 142
杨辉三角问题 143
游程编码问题 145
汉诺塔问题 172
传染病问题 174
迷宫问题 176
八皇后问题 180
文字游戏问题 246
骑士周游问题 249
旅游交通路线问题 260
道路修建问题 277
拼写检查问题 358
犯罪团伙的问题 369

第 1 章

从数据到算法

欢迎学习数据结构与算法。数据结构是计算机科学中的基础理论之一。任何问题解决方案都不可能脱离数据结构而单独存在。本章作为全书的导引章节，将从数据与数据结构的概念谈起，帮助读者对将要学习的内容有一个初步的认识。

1.1 数据与数据结构

数据是一切有意义信息的基本存在形式。如何有效地组织和利用数据，长久以来一直是计算机科学家们思考的问题。无论是学术研究还是工程应用，数据和数据结构都扮演着极其重要的角色。本节主要向读者介绍有关数据和数据结构的一些概念。

1.1.1 数据及其类型

数据（Data）是信息的载体。有序的数据组织就形成了信息。信息是人类可以直接利用或感知的意识形式。而数据则是用来被计算机识别、存储和处理的，它是计算机利用或感知的基本单位。

在计算机科学中，更加具体的数据定义可以表述为计算机加工和处理的对象，这里数据可以是数值数据，也可以是非数值数据。数值数据包括一些整数、实数或复数等，它们主要用于工程和科学计算等；非数值数据包括文字、图像、声音等。在计算机中，这些数据都以二进制数的形式存放在物理介质上。每个二进制位称作一个位（bit），每 8 个二进制位组合起来称作一个字（byte）。

那么这些 0 和 1 是如何组成复杂而有意义的数据的呢？这就需要用到数据类型这个概念了。所谓数据类型就是一个值的集合和定义在这个值集上的一组操作的总称。一般而言，数据类型可分为原子型和结构型。在程序设计语言中，每一个数据都属于某种数据类型。类型明显或隐含地规定了数据的取值范围、存储方式以及允许进行的运算。这些已经事先定义好的数据类型就是所谓的原子型。经过数据类型规定后的若干有序 0 和 1 的组合就在计算机中呈现出一定的意义，最初的表现形式就是原子型数值数据。而在某些时候，一旦需要引入某种新的数据类型时，总是借助编程语言所提供的原子型数据类型来描述或定义新数据的存储结构，这也就是所谓的结构型。换句话说，原子型数据经过一定规则重组后即可形成结构型数据。

那些非数值型的数据也是用 0 和 1 来存储和表示的，只不过规则要复杂得多，其内容已经超出了本书的研究范围，这里不打算具体展开，有兴趣的读者可以参阅本章参考文献[1]以了解更多。下面首先来看看 C++语言中的几种常用原子数据类型，见表 1-1。这些数据在内存中存储时占据了多少内存空间呢？通过下面这段程序可以测得一些结果。

求职面经点拨

请试回答附录题库中的第 1 题。

```
#include <iostream>

using namespace std;

int main(int argc, char** argv) {

   cout << "bool: " << sizeof(bool) << endl;
   cout << "char: " << sizeof(char) << endl;
   cout << "short: " << sizeof(short) << endl;
   cout << "int: " << sizeof(int) << endl;
   cout << "long: " << sizeof(long) << endl;
   cout << "float: " << sizeof(float) << endl;
   cout << "double: " << sizeof(double) << endl;

    return 0;
}
```

求职面经点拨

请试回答附录题库中的第 2 题。

表 1-1　C++语言中的常用原子数据类型

类　型　名	描　述	取 值 范 围
bool	布尔型	true, false
char (signed char)	字符	–128～127
unsigned char	无符号字符	0～255
short (signed short)	短整型	–32768～32767
unsigned short	无符号短整型	0～65535
int (signed int)	整型	–2147483648～2147483647
unsigned int	无符号整型	0～4294967295
long (signed long)	长整型	–2147483648～2147483647
unsigned long	无符号长整型	0～4294967295
float	浮点型	3.4×10^{-38}～3.4×10^{38}　（绝对值精度）
double	双精度浮点型	1.7×10^{-308} ～1.7×10^{308} （绝对值精度）
long double	长双精度浮点型	1.7×10^{-308} ～1.7×10^{308} （绝对值精度）

编译并运行上述程序，结果如图 1-1 所示。可见布尔型数据在内存中仅占 1 字节，字符型数据也占 1 字节；短整型占 2 字节，整型、长整型和浮点型占 4 字节；双精度浮点型占 8 字节。同样一组 0 和 1 的序列如果数据类型不同，那么解释出来的结果将是不同的。

仅有原子型数据仍然是不够的。将原子型数据按照一定规则重组，就可以形成结构型数据。换句话说，结构化的数据可以由简单的数据组成。将某人的联系地址想象成一个结构化的数据，那么在这个结构中可能有若干个项目，它们都是一些简单的基本数据。例如，居住地的门牌号，然后是所在街道的名字、城市的名称、省份的名称和邮

政编码，所有这些都是简单的数字和汉字的组合。

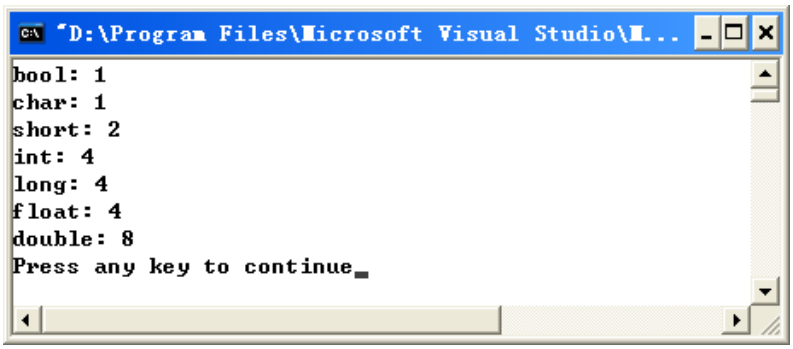


图 1-1　程序运行结果

C 语言允许用户定义自己的数据类型，基本方式是通过结构体的形式将基本数据类型进行组合从而生成结构型数据，而且结构体中的成员仍然可以是结构体类型。C++继承和保留了结构体这种语法，但为了满足面向对象的要求，C++也使这种形式得以进化，它使用类和对象来产生新的数据类型。

一个结构化的数据可以由一些简单的基本数据类型组合而成。而事实上，如果一个结构化的数据包含了其他结构时，它也可能变得非常复杂。例如，一个图书馆有很多书柜，书柜中存放了很多书籍。书柜中的每层格架上可以放一些书，也可以放一些装书的盒子。在这个例子中，书柜是一个由书籍与盒子组成的数据结构。盒子也是一个数据结构，因为盒子里面还放着书。这样一个结构化的数据就又包含了另外一种结构。结构化的数据既提供了一种存储数据的方式，还提供了一种问题解决途径。在图书馆的例子中，结构化的数据不但帮助将书籍有序地组织在一起，而且在数据查询中起到了至关重要的作用。装满文件和文件夹的柜子是一个展现多个结构化的数据如何重组得到一个新结构型数据的典型例子。这里的文件柜就是一个结构化的数据。文件柜由抽屉组成，而抽屉中装有文件夹，文件夹中又装有文件。一旦数据之间通过结构化的方式被组织到一起，那么也就引出了数据结构的概念。这种将原子型数据组织成结构型数据的过程中所使用的模型，就是数据结构。或者说，数据结构就是一个信息的结构化表示，是为了将原子型数据组织成结构型数据而套用的某种组织模型。

1.1.2　数据结构简介

当今世界纷繁复杂，人们每天面对着林林总总的事物，信息从各个方面以多种多样的形态呈现在人们面前。人们所面对的数据量有时是非常巨大的，如果没有一个有效的方法来储存、组织和表示信息，那么信息将变得没有任何价值。想想某个高校的图书馆里的藏书，如果没有图书管理员维护图书管理系统，那么想找到某本书将是多么费劲的一件事啊。

即使面对一组相对较小的信息时，一个结构化的数据表示方法也是必不可少的。假想公共汽车站等车的人不按顺序争先恐后地抢着上车，那么场面将是多么混乱！相反，

如果人们排着队井然有序地逐个上车，情况就好得多了。很明显，这种线性的数据组织方法在信息存储中是具有举足轻重的地位的。

数据结构，简单说来，就是一个信息的结构化表示。无论是图书馆中用以储存书籍的系统还是电影院里排队的人都是数据结构在现实世界中的实例。可以说，数据结构为存储表示数据以及问题解决带来了极大的便利。例如，在图书馆中，数据结构可以帮助人们解决每一本书该被放在哪个特定位置这样的问题；而在电影院买票时，人们以排队等待的形式可以很容易决定下一个该轮到谁。

数据结构也可以被认为是计算机存储、组织数据的方式。关于它的研究由来已久，至今数据结构已经成为计算机程序设计和问题求解的重要工具。通常情况下，正确地使用数据结构可以为高效的问题解决创造可能。数据结构是数据对象，以及存在于该对象的实例和组成实例的数据元素之间的各种联系。这些联系可以通过定义相关的函数来给出。现实世界中的万事万物都是相互联系的信息体，通过对现实世界进行高度抽象而得到的就是数据对象或数据元素。数据对象之间的关系和组织方式称为结构。基本的结构分为两大类，即线性结构和非线性结构。线性结构又包括队列、链表和栈等。非线性结构则包括树和图等。集合结构是独立于线性和非线性之外的一种结构，集合结构中的数据元素除了同属于一种类型外，别无其他关系。

线性结构中元素之间存在一对一关系。其中，最常用的数据组织方式有两种，即顺序式和链式。顺序存储结构是一种最基本的存储表示方法，通常借助于程序设计语言中的数组来实现。顺序存储结构采用把逻辑上相邻的节点存储在物理位置相邻的存储单元里这种方式进行数据组织。如此，节点间的逻辑关系由存储单元的邻接关系来体现。与顺序式不同，链式结构追求更加灵活的数据组织结构。它不要求逻辑上相邻的节点在物理位置上也相邻，节点间的逻辑关系是由附加的指针字段表示的。链式存储结构通常借助于程序设计语言中的指针类型来实现。

作为非线性结构的代表，树结构中元素之间存在一对多关系，而图结构中元素之间存在多对多关系，因此图结构又被称作是网状结构。

数据的运算是在数据结构之上定义的操作。不同的数据结构具有各自不同的操作，即使对于同一类型的数据结构，如果拥有操作的种类和数目不同，即使逻辑结构相同，数据结构能起的作用也大相径庭。不同的数据结构其操作集是不同的，但某些特定的操作则是具有共性的，这些共同的数据操作包括：

- 结构的生成和销毁；
- 在结构中对满足条件的数据元素进行检索；
- 在结构中插入新的数据元素或删除已经存在的数据元素；
- 对数据结构中的数据进行遍历。

在描述数据结构时常采用抽象数据类型（ADT，Abstract Data Type）作为工具。抽象数据类型是对一个数学模型以及定义在该模型上的一组操作的称谓。抽象数据类型实

际上就是对该数据结构的定义，因为它定义了一个数据的逻辑结构以及在此结构上的一组算法。抽象数据类型描述程序处理的实体时，强调的是其本质的特征、其所能完成的功能以及它和外部用户之间交互的接口。以上这些特征体现了抽象数据类型的数据抽象性。

1.2　算法

瑞士计算机科学家尼克劳斯·埃米尔·维尔特（Niklaus Emil Wirth）曾提出一个著名的公式：数据结构+算法=程序。可见算法的重要性。脱离了算法的数据结构是不足以解决和处理问题的。本节将着重讨论有关算法的一些内容。

1.2.1　算法的概念

关于算法的定义，其表述方式有很多种。一种从强调算法跟数据结构关系角度出发的定义是这样描述的：解决一个问题除了要有一个数据结构来表示和组织信息以外，还需要一系列步骤来完成问题的解决过程。操作并使用数据结构来解决问题的一系列步骤称为算法。关于这个定义，我们可以举个简单的例子来加以说明。来考虑先前谈到过的在电影院排队买票的例子。这里有待处理的问题是如何管理一组等待买票的人。那么正在等待的一队人就是一个数据结构。解决这个问题的算法是当售票员将票卖给队列中排在最前面的那个人后，就将这个人从队列中删除，然后将其后面紧挨着的人排到第一位并卖票给他，直到票卖光，或者没有人排队了为止。

也可以从问题解决角度去定义算法，其描述为：算法是有穷规则构成的为解决某一类问题的运算序列。不论从何种角度去描述算法的定义，一些要素对于算法都是必不可少的。首先是输入，任何算法必须先接受一定的数据输入；然后是输出，经过一系列运算规则的处理之后，算法应将结果输出。

小问号

你可以想到多少种求圆周率π的方法?

大多数情况下，解决同一问题的方法可能有很多种。例如，求圆周率π的近似值这个问题，大家能够想出多少种算法呢？事实上，求圆周率π近似值的方法的确有很多。首先可以使用下面这个公式来求出π的近似值。

$$1-\frac{1}{3}+\frac{1}{5}-\frac{1}{7}+\cdots+\frac{(-1)^{n-1}}{2n-1}+\cdots=\frac{\pi}{4}$$

其次，还可以使用所谓的蒙特卡洛（Monte Carlo）法来求π。这是一种基于随机数而设计的方法，它最早起源于 1777 年法国科学家蒲丰（George-Louis Leclerc de Buffon）提出的随机投针法。1805 年，瑞士天文学家沃尔夫（Johann Rudolph Wolf）使用该方法求得的π值为 3.1596。1901 年，意大利数学家马里奥·拉兹瑞尼（Mario Lazzarini）使用该方法求得的π值为 3.1415929。关于原始的蒲丰投针方法这里不再赘述。但需要说明的是原始的蒲丰投针方法是极其繁杂而低效的。随着计算机技术的发展，在计算机上模拟实现蒙特卡洛法求解π值的过程已经十分方便了。众所周知，圆的面积公式为：$S=\pi r^2$，当$r=1$时，$S=\pi$。如果在一个边长为 2 的正方形中有一个内切圆，那

么该圆的面积就为π。利用这一原理，可以假设有大量随机点等概率均匀地落入正方形中。若令S表示内切圆的面积，S'表示正方形的面积，落入圆中的随机点数为N，随机点的总数为N'，这时可以得知$S/S' = N/N'$，所以可据此求得$\pi = 4N/N'$。

可见，在求圆周率π的近似值问题上的确存在着不同的算法。那么这些算法都具有哪些共同的特征呢？通常一个算法应具备以下一些特性。

有穷性：一个算法必须有一定的终止条件来控制处理过程的结束。例如，前面的电影院排队购票算法就规定只要票售完或者没有人买票了，算法就结束。一个需要无穷无尽地算下去的方法事实上意味着该问题应用此方法具有不可解决性，那么一个无法解决问题的方法也就是没有任何价值的。

确定性：算法的每个步骤必须有明确的描述，且不能存在二义性。这也就是保证算法能够被机器执行的必要条件。假想算法的某个步骤缺乏确定性描述，那么如何要求计算机按照程序执行呢？

可行性：算法的可行性表示算法的任何一个步骤都是可以被机器或人执行的。想想中国古代的曹冲称象的故事，曹冲的称象方法就是一个具有可行性的方法。而其他大臣们设想可以把大象切成几块，或者造一个巨大无比的称的方法都是缺乏可行性的。缺乏可行性的程序是不具备实际意义的。

正确性：一个算法的正确性是指这个算法应该能够达到预期的结果。比如曹冲称象的例子，大象的重量最后被测量了出来，那么说明这个方法达到了预期的结果。中国古代典故刻舟求剑、揠苗助长等都是缺乏正确性的方法的典型。

小提示

目标管理中的“黄金准则”——SMART 原则在算法设计与问题求解中具有相通的指导意义。

算法的这些特性也反映了算法设计上应当且必须遵守的一些原则。这些原则和人们日常解决问题时所需要恪守的规则是一致的。现实中设立一个切实可行又经济高效的目标并非一件易事。因为人们经常会犯一些基本的错误，例如过于理想、违反客观规律，或者不易执行等。为了使目标得以实现，且能够正确设计出实现目标的策略，早在 20 世纪 70 年代，美国人就提出了经典的 SMART 原则，它提供了设立目标时所应遵循的简单且容易的规则。这里的 SMART 原则是指：

- **Specific——具体的，明确的**

目标本身不能是模糊的。因此，我们在设计算法时首先要考虑清楚问题本身的目的到底是什么。如果没有真正把握住问题的实际意义和目的，那么算法的设计往往是徒劳的。

- **Measurable——可衡量的**

目标必须是可以衡量的，应当能够让执行者和评估者都清楚地知道衡量的标准。这就涉及算法的分析问题，在 1.2.2 节中我们将讨论算法的分析。

- **Attainable（或者 Achievable）——可实现的**

在目标管理学中，这一条是指绩效指标在付出努力的情况下应当可以被实现，即避免

设立过高或过低的目标。但在算法设计中，如果一个问题已经提出，所谓的可实现性就是指解决该问题所采取的算法应当能够给出最终期待的结果。这对应了算法的可行性特征。

- **Result-oriented（或者 Realistic）——结果导向的**

结果本身是在实现目标前就清晰确立的，且与既定策略的导向性应当一致。反映在算法设计过程中这就意味着，算法局部的每一步或者整体的每一阶段都应该是朝着问题结果的方向推进的，否则就会出现南辕北辙的情况。这对应了算法的正确性特征。

- **Time-limited（或者 Time-based）——有时间限制的或有实效的**

目标是需要在特定的时间内达成的，某一时期内的既定目标如果过了允许的时限就意味着失败。这对应了算法的有穷性特征。作为更高的要求，我们同时还希望算法是能够在尽量短的时间执行完成的，也就是算法的高效性。

这本书既介绍数据结构，又讨论算法，这主要是因为二者是相辅相成、密不可分的。一方面，算法一定要借助相应的数据结构才能得以实现，另一方面我们在定义一个数据结构的同时其实也已经定义了与之相关的操作。这些操作本身执行的步骤就是算法。

小案例

艾派尔求解三体问题的过程是程序优化的经典案例。

神魔小说中，峨嵋派有两件可分可合、威力至大的上古遗珍，雄名紫郢，雌名青索，合称“紫青双剑”。蜀山剑侠若能得此二宝、合而为一，即可无坚不摧、无往不利。而算法与数据结构就如同紫青双剑一样，若二者配合得当，将极大地提升问题求解的效率。历史上一个非常有名的成功案例出现在 1985 年计算机科学家安德鲁·艾派尔（Andrew Appel）发表的一篇文章中。艾派尔正是恰当地运用数据结构与算法，才对原本耗时颇巨的问题进行了成功优化。彼时，艾派尔编写了一个用于计算重力场中天体相互作用的经典物理问题的程序，给定场中物体的质量、初始位置和速度等条件，该程序就可以在三维空间中对这些物体的运行状态进行模拟和仿真。艾派尔在他的论文中讨论了 10000 个天体相互作用时其中两个天体的物理作用问题。之所以说这个优化案例是比较成功的，那是因为艾派尔成功地将本来需要 1 年才能计算完成的程序时间有效地缩短到了 1 天！

我们当然不必费力去考虑那些复杂的物理学问题，也无须对艾派尔程序的具体做法进行深入的讨论和说明。我们所关心的是要实现这样惊人的优化效果，艾派尔到底从哪些方面做了改进？总的来说，艾派尔主要做了 5 个方面的工作。

- **算法和数据结构设计**

艾派尔基于二叉树设计了高效的算法，并以此来提高程序的运行效率，使时间复杂度由最初的$O(n^2)$降低到了$O(n\log n)$。这一改进使程序的运行时间缩短为原来的十二分之一。

- **算法调优**

在得到了基本算法之后，艾派尔又针对具体问题对原算法进行了改进，结果使速度提

高了 2 倍。

- **数据结构改进**

初始数据集被存储在一棵树中，但是随着计算的进行，数据集中的数据会发生一定的变化，这样效率就会慢慢降低。艾派尔通过对每步计算后的数据结构进行重组，从而实现了总运行时间减半。

- **代码优化**

具体的代码优化工作可以分为两类，一类是与系统相关的代码优化工作，另一类是与系统无关的代码优化工作。在与系统相关的代码优化方面，艾派尔发现程序的 98%的计算时间都消耗在了一个函数中，于是他就使用汇编语言重写了该函数，结果效率比原来提高了 2.5 倍。在与系统无关的代码优化工作方面，艾派尔使用单精度浮点数代替了原来的双精度浮点数，结果使程序的效率比原来提高了 2 倍。

- **更换硬件**

艾派尔最初在一台价值 25 万美元的计算机上运行他的程序，结果经过优化之后，该程序仍然需要 2 天的时间才能计算出结果。后来艾派尔选择了一台稍贵一些的计算机，结果程序的运行时间就缩短到了 1 天。

做了以上 5 个方面的工作后，艾派尔成功地将原始程序加速了约 400 倍。当然，效率的提高也是有代价的，原来只需要几十行代码就能完成的程序，最终改进后的代码超过 1200 行，艾派尔从设计到实现这个程序，耗时长达数个月之久！但与原程序要执行 1 年才可得出结果的耗时相比，艾派尔的优化仍然是成功的。而在这一过程中反复优化数据结构和算法无疑是他的制胜关键。

1.2.2 算法的分析

好的算法可以使程序更加出色地解决实际问题。究竟什么算法才是好的算法？评价算法有没有一定的标准？本节将针对以上两个问题展开讨论。

算法的优劣主要是针对算法的效率而言的。一个好的算法应当具有高效性的特点，即算法要尽量节省空间以及时间资源。

首先来定义空间代价和时间代价这两个常用概念。算法在空间上的消耗称为算法的空间代价，指算法在运行过程中消耗的存储空间；算法在时间上的消耗称为算法的时间代价，指算法运行过程所需要的时间。算法的效率与很多因素有关，这决定了测试算法效率的方法有很多种，每种测试方法的侧重点也各不相同。

算法分析技术中最常用的一种称为算法的事前估计。所谓算法的事前估计是指利用 1.2.1 节中介绍的几条标准对算法进行综合评估，但这种评估通常在算法被调用前进行，所以称之为事前估计。事前估计通常用来分析算法的可行性，即算法究竟有多大的利用价值。举个例子来说，如果一个算法能够很好地完成预定功能，但这种完成是基于

小数据量的，一旦算法遇到稍微大一些的数据量，它便要花费巨大的时间以及空间资源，那么这种算法便不适于解决数据量庞大的问题。需要说明的是算法的事前估计仅仅是从某些定性角度分析算法，它并不能给出精确的定量结果。

事前估计分为对算法空间复杂度的度量与对算法时间复杂度的度量。空间复杂度是指当问题的规模以某种单位从1增加到n时，解决这个问题的算法在执行时所占用的存储空间以某种单位由1增加到$s(n)$，则称此算法的空间复杂度为$s(n)$。时间复杂度是指当问题的规模以某种单位从1增加到n时，解决这个问题的算法在执行时所耗费的时间以某种单位由1增加到$t(n)$，则称此算法的时间复杂度为$t(n)$。

值得注意的一点是，对算法进行空间复杂度以及时间复杂度的估计通常得到的只是算法在空间代价及时间代价方面的近似值，并不是准确值。这是因为准确值的获得将使估计过程十分烦琐，也没有必要。举例来说，若经过准确估计后得到$t(n)=n^5+n^3+n$，则当n很大时，$t(n)$的值近似等于n^5，所以在对原算法进行估计时，直接估计算法中时间复杂度最大的部分就可以了。这称为对时间复杂度的渐进表示法。类似地，还有对空间复杂度的渐进表示法。

利用渐进表示法得到的算法复杂度通常用字母O表示，称为大O表示法。例如，上面例子中算法的时间复杂度最终由n^5项决定，则称算法的时间复杂度为$O(n^5)$。易见，大O表示法得到的是算法复杂度的数量级信息，这个数量级信息是程序在最坏情况下的时间代价。至于什么是程序的最坏情况需要视具体程序而定，读者不必在此深究，下文中将结合代码介绍程序的最坏情况这个概念。

使用大O表示法时有所谓的加法规则与乘法规则，现分述如下。

大O表示法的加法规则是指当两个并列的程序段的时间代价分别为$O[T_1(n)]$与$O[T_2(n)]$时，那么将两个程序段连在一起后整个程序段的时间代价为：

$$O[T(n)]=O\{\max[T_1(n),T_2(n)]\}$$

大O表示法的乘法规则是指当两个嵌套的程序段的时间代价分别为$O[T_1(n)]$与$O[T_2(n)]$时，那么整个程序段的时间代价为：

$$O[T(n)]=O[T_1(n)\times T_2(n)]$$

使用大O表示法估计程序复杂度只需要考虑程序关键操作的步数，而无须考虑非关键操作对算法复杂度造成的影响。究竟什么是程序的关键操作呢？关键操作大多数存在于循环与递归中。由于递归的概念在后面的章节中才会介绍，所以这里只探讨循环语句中关键操作的问题。

对于单个循环而言，在循环内的简单语句即为关键操作；对于多个并列的循环而言，先分别计算各个循环的算法复杂度，最后再利用大O表示法的加法规则计算整个并列循环的算法复杂度。

利用大O表示法既可以计算算法的时间复杂度，又可以计算算法的空间复杂度，下面

将分两部分来介绍这两种算法复杂度的计算方法。

1．算法的空间复杂度

根据算法在执行过程中对存储空间的分配方式可以把存储空间分成两种：静态空间与动态空间。静态空间主要包括存放程序代码的空间，常数、数组，以及对象的数据成员等所占用的内存空间。动态空间是程序在运行过程中动态开辟的存储空间，例如，链式栈在实现过程中要不断使用 new 语句创建新节点，然后再把新节点加入栈中。使用 new 语句开辟的内存空间在程序运行结束前要调用 delete 语句进行释放，以防止内存泄漏。

对静态空间进行空间复杂度分析比较简单，只需弄清楚算法在执行过程中使用的所有变量，再利用相应的存储空间单位计算出程序需要多少静态空间存储这些变量即可。例如下面的代码。

```
int a[SIZE];
for(int i=0;i<SIZE;i++)
{
    a[i] = i;
}
```

程序使用了长度为 SIZE 的一维数组存储整数，由于在程序的执行过程中数组a的大小不变，所以数组a存放在静态存储区中。程序的空间复杂度为$O(n)$。

对动态空间进行空间复杂度分析比较困难。如果某段程序调用了x次new语句和0次delete语句，则此程序的空间复杂度为x。若某段程序在调用new语句的同时也调用delete语句，假设delete语句的调用次数为y，则整个算法的空间复杂度并不简单地等于$x-y$，而是要根据具体情况具体分析。分析方式为把程序中new语句与delete语句转化为序列，然后再分析这个序列的复杂度即可。限于本书所讨论的内容，有关动态空间的复杂度估计问题此处将不做深入讨论，有兴趣的读者可以查阅相关资料以进一步加以理解。

2．算法的时间复杂度

对算法的时间复杂度进行分析有两种方式。一种方式是计算算法具体的执行时间，最终根据这个时间的长短评价算法的优劣。这种估计方法比较简单，而且易于操作。但这种估计方法有一个缺点：估计时间会随着计算机的不同而产生变化。不同的计算机之间存在配置上的差别，即使是同一台计算机也会因温度、湿度等环境差异而表现出不同的运算速度。同一个算法在不同计算机之间运行所花费的时间并不相同，这决定了用计算算法具体执行时间的方式不能有效地对算法时间复杂度进行估计。

另一种计算算法时间复杂度的方式是只记录算法的关键操作。有关关键操作的概念前文已介绍得很清楚，这里不再赘述。下面将举实例说明这种估计方法的计算过程。

```
for(int a=0;a<num1;a++){
    for(int b=0;b<num2;b++){
        num[a][b] = b;
    }
```

```
}
```

上面的代码段出现了循环语句的嵌套。在前面曾介绍过，当两个循环语句相互嵌套时算法的时间复杂度的计算要应用乘法规则。上述代码中，$T_1(n) = \text{num1}$、$T_2(n) = \text{num2}$，所以$O[T(n)] = O(\text{num1} \times \text{num2})$。

循环语句的循环条件会对计算算法时间复杂度产生一定的迷惑作用。很多初学者认为循环次数越少的算法在时间开销方面越小。从原理上说这是对的，但不要忘记，大O表示法计算的只是算法在时间开销方面的数量级，若数量级相同，即使两个算法执行关键操作的步数实际上并不相同，那么也把它们看作是时间复杂度相等的两个算法。考虑下面的代码。

```
for(int a=0;a<num1/2;a++){
    for(int b=0;b<num2/2;b++){
        num[a][b] = b;
    }
}
```

上述代码与前面第 1 个例子中的代码在结构上一模一样，唯一的不同之处在于两重循环的循环次数均减少为原来的一半。有读者可能认为这段代码时间复杂度要比第 1 个例子中代码的时间复杂度低，因为执行它需要更少的时间。这种推断对不对呢？如果用公式计算上述代码的时间复杂度可知$O[T(n)] = O(\text{num1} \times \text{num2}/4)$，显然$\text{num1} \times \text{num2}$与$\text{num1} \times \text{num2}/4$具有相同的数量级，两者均为$n^2$，所以上述代码与前面第 1 个例子中的代码具有相同的时间复杂度，虽然这段代码关键操作要少很多。

下面再来看一段代码，请读者试着分析这段代码的时间复杂度为多少。

```
Node* Find(const Type& x ,Node* ptr){
    if(ptr==NULL){
        reutrn NULL;
    }
    else if(x<ptr->data) return Find(x,ptr->leftChild);
        else if(x>ptr->data) return Find(x,ptr->rightChild);
            else return ptr;
}
```

可以看到上诉代码定义了函数 Find()。但在实现部分，函数 Find()又调用了其自身。像这种用自身定义自身的情况就叫作递归。函数 Find()的作用是在指针ptr指向的链表中寻找变量x，并返回指向x所在节点的指针。这里链表中的数是有序存放的，现假定它为从小到大的顺序。Find()的执行过程是这样的：首先令x与链表中间的数值进行比较，如相等，则返回节点指针；若小于，则在中间节点左侧的链表中寻找变量x，寻找方式与上面相同；若大于，则在中间节点的右侧链表中寻找变量x，寻找方式与上面相同。如此递归下去直到算法结束。上述查找法叫作折半查找法，它是比较常用的一种查找法。由于采用折半的方式，若链表的长度为n，则平均查找的时间为$\log_2 n$，所以算法的时间复杂度为$O(\log_2 n)$。

常见的程序时间复杂度有（按照从小到大的顺序排列）：c、$\log_2 n$、n、$n\log_2 n$、n^2、n^3、2^n、3^n、$n!$，图 1-2 是其中几种大小关系示意曲线。其中c表示常数，即算法的执行时间是固定不变的，不随程序中其他变量状态的改变而发生变化。但要注意，语句

“a = 1;”与语句“a = (a + b) * c + (d + e) * f + g/h;”的时间复杂度同为c，不可因为后面的赋值语句较长就认为它具有更高的时间复杂度。

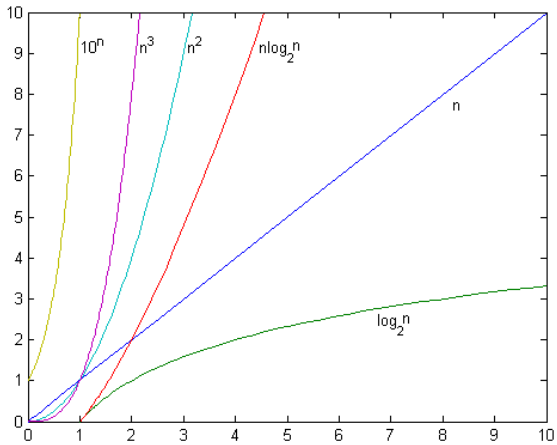


图 1-2　常见程序复杂度

在设计算法时要尽量减少算法的时间复杂度。若程序可以用时间复杂度为n的算法实现就避免使用时间复杂度为n^2的算法。但时间复杂度并不是衡量程序的唯一标准。在注重时间复杂度的同时还要注重算法的正确性及可读性等其他特性。

1.2.3　算法的设计

本节介绍算法设计方面的知识。现实生活中存在着各种各样的问题，如何利用计算机有效地解决这些问题是许多科学家多年来关注的焦点。由于问题的多样性，不可能有一种算法“通吃”所有问题，即使是再优秀的算法也有一定的适用范围。所以，多掌握一些算法设计思想才能使我们灵活地应用编程语言对问题进行求解，才能在面对复杂问题时不至于没有思路，无从下手。本节所介绍内容更准确地讲并非是具体的某个算法，而是算法设计的经典思想。本书后面章节中所介绍的诸多算法则是这些思想的具体化。随着学习的深入，读者将会不断深化对这些思想的认识和理解。

1．贪婪算法

什么是贪婪算法呢？下面举一个现实当中的例子。假设一件商品值 23 元 4 角，现在顾客给售货员 100 元钱，问题是，售货员如何用最少的纸币给顾客找出特定金额的零钱。在这个问题中，售货员要找给顾客 76 元 6 角的零钱，最好的方法是给顾客一张 50 元的纸币，一张 20 元的，一张 5 元的，一张 1 元的，一张 5 角的以及一张 1 角的。只有用这种方法才能保证找给顾客的零钱由最少的纸币组成。大部分人在日常生活中都用到类似的方法来解决找零钱的问题，每一次选出最大面值的，并且和已选出的纸币金额加起来不超过应付总金额的纸币找给对方。这种方法就叫作贪婪算法。

可用贪婪算法解决的问题有如下几个特征。

- 该问题至少有一个最优的解决方案。只有这样，才可能为解决问题构造候选方案集合。在前面的例子中候选方案集合就是不同面值的纸币组成的纸币集。
- 随着问题的求解将建立另外两个集合，一个集合（假设为集合a）由已经被选出并符合问题要求的候选解组成，另一个集合（假设为集合b）由已经被选出但并不符合问题要求的候选解组成。
- 集合a最终要包含问题的解答。在集合a中未出现解答时，问题的候选解集合中至少还要剩余一个候选解。

利用贪婪算法可解决的经典问题有：

- 图的最小生成树问题（克鲁斯卡尔算法及普里姆算法）。
- 图的最短路径问题（狄克斯特拉算法）。
- 最小化任务执行的平均时间问题。
- 背包问题。

贪婪算法的优点是简单直观，易于实现。它的缺点是在解决问题的策略上目光短浅。这种算法每当遇到满足问题条件的候选解时就认为这个解是正确的，毫不关心这个解对问题后续求解的影响。因此利用贪婪算法解决的问题是十分有限的。

2．分治法

什么是分治法呢？请看下面这个简单的例子。现在有一个问题，要计算123×321的值是多少。有的读者可能马上给出答案39483。是的，这就是问题的正确解答，而求解这个问题的算法也很简单，只需要利用乘法运算法则就可以了。但是，乘法运算中却蕴含着本节要讨论的分治法的思想。

在计算乘法时，总是把其中的一个乘数拆分成几部分，然后用这几部分与另一个乘数相乘，最后再把得到的积相加作为整个乘法运算的积。例如上面的乘法，首先把321拆分成$321 = 3 \times 10^2 + 2 \times 10^1 + 1 \times 10^0$，然后再用这个拆分后的数与123相乘，即

$$1 \times 10^0 \times 123 = 123$$

$$2 \times 10^1 \times 123 = 2460$$

$$3 \times 10^2 \times 123 = 36900$$

最后把以上乘法的结果123、2460、36900相加就可得到123×321的积39483。可见在计算乘法时并不是用两个乘数直接相乘，而是把其中的一个乘数进行“ 分解 ”，这种分解求解的方法就叫作分治法。

可用分治法解决的问题有如下几个特征。

- 整个问题可以分解为两个或多个较小的子问题。
- 对每个子问题的求解类似于对整个问题的求解。

- 如果子问题的规模仍相当大而不可以对子问题方便求解，那么再对子问题应用分治法。

采用分治法对问题进行求解有一个通用的模板，现把这个模板描述如下。

```
function DivideToConquer(x){
    if (x足够小且足够简单) {处理x, return }
    else{
        把问题分解成子问题 x1, x2, x3…xn
        for(int i=0;i<n;i++){
            处理xi , 即DivideToConquer(xi);
        }
}
```

注意以上模板只是用分治法处理问题的一般模型，读者要好好体会模型的实质，牢牢掌握分治法的精髓。切不可形而上学地利用上述模板生硬地编写代码，或是直接把问题的条件带入模板中，更不能直接编译上述模板。

求职面经点拨

请试回答附录题库中的第3题。请注意体会其中的分治思想。

利用分治法解决的经典问题有：

- 传染病问题。
- 二分搜索及快速排序。
- 矩阵乘法。
- 汉诺塔问题。

分治法的应用范围很广泛，很多很复杂的问题利用它都可以得到清晰快速的解决。但分治法要比其他的算法更加难以理解，难以把握，除非对问题有精辟透彻的分析，否则难以应用分治法解决问题。这是因为分治法要求对原问题进行分解，而如何对原问题进行分解、分解到什么程度才合适需要视原问题而定，这便是应用分治法的难点所在。

分治法的一个缺点是效率不高，若对原问题进行不恰当的分解会进一步降低分治法的效率。总之，分治法是一种技巧性很高的算法，若利用不当不但不会提高解决问题的速度，还会使求解问题的过程变得异常复杂。关于分治法的更详细内容在本书的后续章节中还会被讨论。

小提示

本书后面还会介绍许多基于动态规划思想设计的经典算法，例如：德勒曼-温施算法以及基于动态规划的最短路径算法等。

3．动态规划

分治法是一种自顶向下（top-down）的算法，它把问题划分成一个一个的子问题，通过对子问题的求解实现对原问题的求解。动态规划与分治法的思想相反，它是一种自底向上（bottom-up）的设计方法。动态规划的思想方法是避免对同样的问题进行两次求解。快速傅里叶变换的计算过程就用到了动态规划的思想。

为了使读者对动态规划有一个清晰的概念，下面举一个例子来说明什么是动态规划。在计算二项式系数时可以利用前面讲到的分治法，如计算C_m^n可使$C_m^n = C_{m-1}^n + C_{m-1}^{n-1}$，其中$m > n > 0$。然后再分别计算$C_{m-1}^n$及$C_{m-1}^{n-1}$的值。若它们还可再分，则对$C_{m-1}^n$及$C_{m-1}^{n-1}$

再应用一次分治法，直到不可再分为止。最后把分治法得到的所有结果相加便是二项式系数C_m^n的值。

上述算法似乎可以有效地计算任何一个二项式系数，而且算法用到了分治法的思想，使得实现程序简单清晰。那么它是否可以称得上一个好算法呢？有没有比它更优秀、更便捷的算法呢？通过分析上述算法的递归树可知它的效率是很低的，因为它重复计算了很多已经计算过的值。图 1-3 显示了计算C_5^3时所用到的递归树。

由图 1-3 可见，在计算C_5^3时递归算法重复计算了3次C_1^0、C_1^1及C_2^1。由此可见分治算法的效率是很低的。当C_m^n中的m与n取值较大时分治算法将计算更多的重复项。

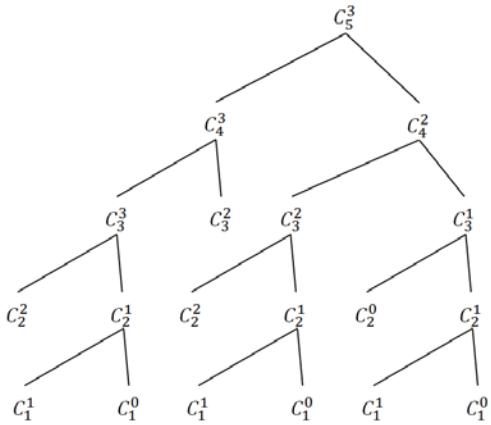


图 1-3　计算C_5^3时所用到的递归树

计算二项式系数的另一个方法是利用杨辉三角。利用它不需要每次计算新的数值，只需要根据三角形中已存在的数值就可以构造出新的二项式系数。这种思想就是动态规划的思想。由于减少了重复的工作，算法的时间复杂度也将显著降低。

利用动态规划思想解决的问题有如下几个特征。

- 算法要解决大量重复的工作，计算大量重复的数值。
- 至少有一些新的数值可以通过已经计算出的数值组合出来。
- 问题要具有一定的规模，这样动态规划算法的好处才可凸显出来。
- 使用分治法划分子问题时，子问题之间的关系界限不清，相互交叉。一个子问题的求解与多个已经解决的子问题有关。

动态规划法虽然可以提高程序运行的效率，但它是以牺牲算法的可读性为代价的。通常利用动态规划法写出的算法均比较复杂，因此降低了算法的可读性。动态规划法的另一个缺点是它有时不得不解决一些无关的子问题，因为动态规划法无法知道以后是否会需要这些子问题的解。以上两个缺点是分治法所没有的。可见动态规划法与分治法各有所长。读者可根据实际情况适当地选用这两种方法。

动态规划法与分治法的最大区别在于动态规划法所划分的子问题之间联系比较密切，

子问题与子问题的界限不是很清楚，往往在解决了一个子问题的同时也在某种程度上解决了另一个子问题。利用动态规划法处理问题遵循自底向上的顺序，子问题数目较多。

而分治法所划分的子问题之间联系甚少，几乎没有关联。子问题与子问题之间界限清楚明了，解决一个子问题不会为解决另外的子问题造成任何影响。利用分治法处理问题遵循自顶向下的顺序，子问题数目较少。

利用动态规划法解决的经典问题有：

- 快速傅里叶算法。
- 多边形的最优分割。
- 找零钱问题。
- 最短路径问题。
- 背包问题。
- 链式矩阵乘法。

在上述经典问题中有几个可以利用前面介绍的算法求解。如找零钱问题、最短路径问题以及背包问题均可利用贪婪算法解决。由此可见好的算法是具有极强的灵活性的，它不能只解决一类问题，而是要体现一定的“通用性”。

4．回溯法

回溯法是一种优化的穷举法。所谓穷举法就是穷举问题的所有可能性，直到发现解决问题的最佳解为止。当然利用穷举法可以解决很大一部分的实际问题，但由穷举法的实现过程可知它所花费的资源是相当多的。使用穷举法进行彻底的搜索、彻底的比较，要以大量的运算为基础，而大量的运算势必要花费大量的时间，有时这种消耗是计算机也承担不起的。所以用穷举法解决问题往往是不实际的。

回溯法会有效地降低穷举法带来的高损耗。下面举一个例子来说明什么是回溯法。考虑对迷宫进行遍历。当遇到十字路口时究竟应当向什么方向走呢？回溯法为我们提供了一个答案。每当遇到十字路口时便向一个固定的方向前进。在这里假设这个方向是左。当向左走后如果又遇到了十字路口，那么便继续向左走；如果遇到了死胡同便回溯到前一个路口并直走（因为此路口的左边已被证实是走不通的）。当直走并遇到路口后继续向左走，当直走并遇到死胡同后同样回溯到前一个路口（此时这个路口就只剩下右边可走了）。当向右走后如果遇到路口仍然向左走，当向右走并遇到死胡同后仍然回溯到前一个路口。此时这个路口已被证实是走不通的，于是再向前回溯一个路口。如此循环下去直到找到迷宫的出口。关于回溯法和迷宫问题的具体求解过程在本书的后续章节中将会有更为详细的描述。

以上便是回溯法的基本思想。利用回溯法解决的问题有如下几个特点。

- 问题可以划分为几个不相关的子问题。
- 几个子问题有类似的求解方式。
- 如果子问题还可以划分，便继续对子问题进行划分。

易见，利用回溯法解决的问题与利用分治法解决的问题有相似之处。但回溯法与分治法并不相同。回溯法的本质是深度优先搜索，在应用回溯法时需建立一个栈保存搜索路径。由栈的工作方式可知当前的搜索路径一定保存在栈的顶部，当该搜索路径被确定为不可行时便从栈顶删除此搜索路径并对新栈顶所保存的搜索路径进行搜索，如此循环下去直到找到问题的解或栈为空栈。而分治法的实质是递归。分治法每次解决整体问题中的一个子问题，子问题可能被重复解决，但不会出现像回溯法中那样走进“死胡同”的情形。

利用回溯法能解决的经典问题有：

- 迷宫问题。
- 八皇后问题。
- 骑士周游问题。
- 图的深度优先遍历。

回溯法的一个缺点是效率较低。当前求解的路径也许是错误的路径，这样回溯法就要放弃当前的路径转而选择一条新的求解路径。而新的求解路径在对其计算之前是不能确定正确性的。

5．概率算法

与确定性算法相比，概率算法不能得到精确的结果。但这并不意味着概率算法得不到正确的结果，而且在某些场合概率算法得到正确结果的效率要比精确算法高很多。下面就是一个应用概率算法的例子。

如果一个国家要选举一位总统，而且假设这个国家所有公民都要参与选举过程，那么如何在尽可能短的时间内选出最合民心的总统呢？这里有两种选举方式。一种选举方式为全民投票选举，向所有的公民发放选票，最后再把这些选票集中起来进行统计，统计的结果表明哪位候选人得到的票数最多哪位候选人便当选为总统。这种选举方式可以保证得到精确的选举结果，并且符合要求的总统最终只有一个。但这种方式的最大缺点是选举周期过长，选举过程烦琐。如果一个国家的人口总数很多（比如说 1 亿公民），那么这个国家就需要相当长的时间来选举一位总统。这显然是不合适的。

另一种选举方式是民意测验法。在这种选举方式中不向选民发放选票，而是向他们发放民意调查表，调查这些总统候选人在选民当中的支持率。为了减少选举时间，可以向每 10 个或 20 个选民发放一张民意调查表，最后把这些民意调查表汇总起来进行统计，哪位候选人的支持率最高哪位候选人就被选举为总统。

民意测验法的优点是周期短，且结果符合绝大多数选民的意愿。缺点是不够精确。但在选举总统这个例子中，由于最终总统只能有一人，所以总有一部分选民对选举结果不满。这样，只要选举出的总统符合大多数选民的意愿就认为选举的结果是合理的。在这个前提下民意测验法相对于投票法来说是一种又快又“准”的选举法。

民意测验法就是应用概率算法解决实际问题的一个例子。概率算法的实质是算法对于同样的输入产生不同的结果。如果这个结果经检验是正确的，那么算法结束；如果这个结果经检验是不正确的，则算法放弃该结果，从头再进行一次计算。对于精确计算的算法来说，如果输入错误的参数，那么算法总是得到错误的结果；而对于概率算法而言，如果输入错误的参数，那么算法可能得到错误的结果。如果现实问题的条件限定发生错误结果的概率是很小的，那么概率算法的优点就会凸显出来。

概率算法的另一个好处是有可能得到多个正确答案，这个特性使得概率算法在求解最优解问题时有很好的性能。精确算法只能得到一个最优解，这个最优解是否能被应用到实际中还需进一步的检验。如果这个最优解被确认为是没有实际意义的，那么程序员就要修改算法，让算法能求解次优解。而应用概率算法可以有效地解决计算次优解的问题。由于概率算法所得到的结果只是对正确结果的一个近似，所以当算法结束时很可能在计算出最优解的同时产生了多个次优解。若最优解不适用于实际情况，程序员不必修改算法，只需检验与最优解一同产生的次优解是否可应用于实际就行了。

由于概率算法涉及很多概率论与数理统计方面的知识，限于本书所讨论的内容，在这里仅限于对这种算法做一简单的介绍。有兴趣的读者可以查找相关资料，以了解更多。

经典的概率算法有蒙特卡洛（Monte Carlo）算法和拉斯维加斯（Las Vegas）算法。应用概率算法可解决的经典问题有：

- 蒲丰投针问题。
- 验证矩阵乘法。
- 素数性测试。
- 大整数分解因式。

以上介绍了算法的分析与设计技术。这两种技术在实际中有很广泛的应用，而且只有把这两种技术结合起来才能设计出好的算法。对于本节中所提到的算法设计思想读者如感到不是很理解，可以不必在此深究，在本书的后续章节将结合实例详细地对它们进行讨论。本节的目的只是为了让读者对这些算法有一个初步的认识，明白这些算法的应用领域。若想真正掌握好这些算法并在实际中灵活应用它们，读者还需在阅读本书的同时多多历练。

1.3 C++中的 STL

C++是一种极富魅力的语言，在世界范围内有着广泛的用户基础。作为广大程序员最

青睐的计算机程序设计语言之一，C++自产生之初到今时今日历经近 30 年时间风采不减，依然呈现出勃勃生机。标准模板库（STL）是 C++中的一个重要特性，也是本书介绍的重点内容之一。本书之所以会对 STL 进行介绍的原因在于其中的容器与数据结构之间存在某些联系，笔者希望通过介绍 STL 中的容器简化读者在编写一些算法程序时的工作。但需要说明的是，本书不是一本专门介绍 STL 的书籍，因此后续内容中提到的有关 STL 的内容仅限于对其容器的一些介绍。

1.3.1　STL 简介

标准模板库（STL，Standard Template Library）最初是由惠普实验室开发的一系列软件的统称。它是由亚历山大 · 斯特潘诺夫（Alexander Stepanov）等人在惠普实验室工作时所开发出来的。虽然，STL 目前主要出现在 C++中，但事实上在被引入 C++之前该技术就已经存在了很长的一段时间。STL 的出现可谓是一个极其令人兴奋的事件，特别当它与 C++结合到一起并成为 C++密不可分的一部分时，就注定它要成为备受注目的焦点。

STL 的历史可以追溯到 20 世纪 70 年代，那时被誉为 STL 之父的斯特潘诺夫希望能够找到一种在保证效率的前提下，将算法从诸多具体应用之中抽象出来的方法。该思想后来演化成了被称为泛型化思想的概念。为了探索合适的途径，斯特潘诺夫曾经做过很多尝试，其间他和纽约州立大学教授迪帕克 · 卡普尔（Deepak Kapur），伦塞里尔技术学院教授大卫 · 马瑟（David Musser）共同开发了一种叫作 Tecton 的语言。可惜这次尝试最终没有取得实质性的成果。但该过程却给斯特潘诺夫带来了很大的启示，并成为 STL 诞生的起点。该项实验失败后，斯特潘诺夫又和马瑟等人先后用其他一些语言（主要包括 Schema 语言和 Ada 语言等）建立了一些大型程序库。在这个过程中斯特潘诺夫发现了许多问题，于是构建 STL 的思想开始出现了一个朦胧的轮廓。

1987 年左右，在贝尔实验室工作的斯特潘诺夫开始采用 C++语言进行泛型软件库的研究。这是 STL 与 C++结合的最初尝试。不幸的是，当时 C++语言还没有引入模板语法。前面曾经讲过模板的概念，读者很容易想到如果没有模板语法的支持，实现泛型化是多么困难。所以 STL 似乎还没有迎来合适的出现契机。但在这个过程中，与本贾尼 · 斯特劳斯特卢普（Bjarne Stroustrup）和安德鲁 · 克尼格（Andrew Koenig）等顶级大师的共事经历带给了斯特潘诺夫很多启示（前者是 C++语言的创始人，后者是前 ISO C++标准化委员会主席），他开始注意到 C/C++语言在实现其泛型思想方面所具有的潜在优势。次年，斯特潘诺夫开始进入惠普的帕罗奥图（Palo Alto）实验室工作，从那一刻起就注定在该实验室中要发生一场技术革命。

经过长时间的努力，斯特潘诺夫和另外一个研究人员李萌（Meng Lee）完成了一个包含有大量数据结构和算法部件的庞大运行库，这便是现在的 STL 雏形。可以认为是 STL 的真正诞生，他们共同完成的 STL 实现版本，也是 STL 的一个实现版本，后来被称为 HP STL。

1994 年 3 月，斯特潘诺夫正式向 ISO C++标准化委员会提交了一份建议书，以期使 STL

成为 C++标准库的一部分。但当时该建议并没有被马上通过，经过投票表决，委员会决定推迟对该建议的决定。随后在众人的帮助之下，斯特潘诺夫又对 STL 进行了改进，这些改进使 STL 的大部分实现都可以独立于具体平台，从而达到了一个更高的“泛化”层次。在同年夏季的滑铁卢会议上，委员们以 80%赞成，20%反对，最终通过了斯特潘诺夫的提案，决定将 STL 正式纳入 C++标准化进程之中，从此，STL 正式成为了 C++的一部分。

目前，STL 是一个具有工业强度的、高效的 C++程序库。它被容纳于 C++标准程序库中，是 ANSI/ISO C++标准中最新的也是极具革命性的一部分。该库包含了诸多在计算机科学领域里常用的基本数据结构和基本算法。为广大 C++程序员们提供了一个可扩展的应用框架，高度体现了软件的可复用性。

标准模板库体现了泛型化程序设计的思想（generic programming），并引入了诸多新的名词，比如像需求（requirements）、概念（concept）、模型（model）、容器（container）和迭代器（iterator）等。这些都是支持泛型思想的概念，泛型也是一种软件的复用技术，这看起来与 C++的特性是如此相合。

1.3.2 STL 构成

了解 STL 的构成可以很系统化地认识其内部组织，为日后进一步学习 STL 提供有力帮助，总的来说，STL 的构成可以概括为：3 大主体、6 大组件、13 个头文件。所谓 13 个头文件是指在 C++标准中，STL 被组织为下面的 13 个头文件，包括“algorithm”、“deque”、“functional”、“iterator”、“vector”、“list”、“map”、“memory”、“numeric”、“queue”、“set”、“stack”和“utility”。这些头文件包含了全部的代码，而这些代码从广义上讲可以被分为 3 大类：container（容器）、algorithm（算法）和 iterator（迭代器），且几乎所有的代码都采用模板类和模板函数地方式，显然这有利于代码的重用。这 3 类也就是 STL 的 3 大主体。事实上如果细致地考虑 STL 的组成，它应当包括 6 个组件，除了前面比较重要的 3 大主体以外，还包括：仿函数、适配器和空间配制器。下面主要对 3 大主体进行一个概要介绍。

1．容器

容器无疑是 STL 中最吸引人的地方。本书对于 STL 的介绍也主要集中在容器方面，因为 STL 中的容器与本书的主题“数据结构”实在有太多联系。在实际的开发过程中，数据结构本身的重要性不会逊于操作于数据结构的算法的重要性，当程序中存在着对时间要求很高的部分时，数据结构的选择就显得更加重要。这点在本书后面的很多算法题目的解答上表现得尤为明显。

经典的数据结构是有限的，例如本书将要介绍的链表、堆、栈和队列等，但是很多情况下程序员不得不重复着一些为了实现上述有限数据结构而编写的代码，且这些代码都十分相似，只是在细节上有所差别。

STL 容器就为程序员复用这些数据结构提供了方便。STL 容器对最常用的数据结构提

供了支持，这些模板的参数允许程序员指定容器中元素的数据类型，从而使许多重复工作得以简化。此外，程序员还可以重复利用已有的实现构造自己的特定类型下的数据结构，在本书中也提供了这方面的例子。到时，读者可以很容易体会到该项功能的便利性。STL 容器部分主要由头文件“ vector ”、“ list ”、“ deque ”、“ set ”、“ map ”、“ stack ”和 “ queue ” 组成，其中提供的容器主要包括：

- 向量（vector）
- 链表（list）
- 栈（stack）
- 队列（queue）
- 优先队列（priority_queue）
- 双向队列（deque）
- 集合（set）
- 多重集合（multiset）
- 映射（map）
- 多重映射（multimap）

2．算法

算法是 STL 的一个重要组成部分，STL 中大约包含了 70 个通用算法，用于操控各种容器，同时也可以操控内建数组。所有这些操作都是在保证执行效率的前提下进行的，且所有算法都遵循所谓的泛型算法通则，即：

- 所有算法的前两个参数都是一对 iterator：(first,last)，用来指出容器内一个范围内的元素。
- 每个算法的声明中，都表现出它所需要的最低层次的 iterator 类型。
- 大部分算法都可以用仿函数（function object，又称 functor）来更改准则。

STL 利用模板机制提供了相当多的有用算法，其中常用的包括比较、交换、查找、遍历、复制、修改、移除、反转、排序、合并等。如此，在熟悉了 STL 之后，许多代码都可以被大大地化简，只需要通过调用一两个算法模板，就可以完成所需要的功能并大大地提升效率。

STL 中的算法部分主要由头文件 “ algorithm ”、“ numeric ” 和 “ functional ” 组成。头文件 “ algorithm ” 是所有 STL 头文件中最大的一个，也是 STL 中算法部分的主体。它由很多模板函数组成，可以认为每个函数在很大程度上都是独立的。“ algorithm ” 囊括的算法包括查找、交换和排序等。“ numeric ” 体积很小，只包括几个在序列上面进行简单数学运算的模板函数，包括加法和乘法在序列上的一些操作。“ functional ” 中则定

义了一些模板类，用以声明函数对象。

3．迭代器

要完全理解迭代器的含义并不是一件容易的事情。迭代器起初是设计模式中的一种，它的意思是提供一种方法，按顺序访问某个容器所含的各个元素，而无须暴露该容器的内部表述方法。这也是软件设计的一个基本原则，也就是说通过引进一个间接层来简化所有问题的解决。在 STL 中，迭代器被用来将算法和容器联系起来，起着某种黏合作用。容器提供迭代器，而算法使用迭代器。几乎 STL 提供的所有算法都是通过迭代器存取元素序列进行工作的，每一个容器都定义了其本身所专有的迭代器，用以存取容器中的元素。迭代器能够使容器与算法不干扰地相互发展，最后又能无间隙地黏合起来。特别地，迭代器还重载了*、++、==、!=、=等运算符，用以操作复杂的数据结构。

迭代器部分主要由头文件“ utility ”、“ iterator ”和“ memory ”组成。其中，“ utility ”是一个很小的头文件，它包括了贯穿使用在 STL 中的几个模板的声明，“ iterator ”中提供了迭代器使用的许多方法，可以认为是迭代器部分的主体。“ memory ”则以不同寻常的方式为容器中的元素分配存储空间，同时也为某些算法执行期间产生的临时对象提供机制。

1.3.3 STL 的不同版本

ANSI/ISO C++文件中的标准模板库仅是一个被描述在纸上的标准，对于诸多 C++编译器而言，各自使用的 STL 不尽相同，每个不同版本的 STL 都只不过或多或少地实现了标准中所描述的内容，但不同版本之间仍然是存在差异的。本节介绍其中几个比较常见的 STL 实现版本。

1．HP STL

HP STL 是所有其他 STL 实现版本的源头，我们曾经在前面提到过它。该版本是斯特潘诺夫在惠普实验室工作时，和他的同事共同完成的，是第一个 STL 的实现版本。这个 STL 是完全开源的，所以它允许任何人免费使用、复制、修改和发布，前提是必须在所有相关文件中加入 HP STL 的版本信息和授权信息。很可惜在日新月异的计算机科学领域里，早期出现的东西在历史舞台上停留的时间往往是很短暂的，现在 HP STL 已经不再被使用了。

2．Plauger STL

Plauger STL 是 HP STL 的一个继承版本，因此在其所有头文件中都含有 HP STL 的相关声明。它是由飞利浦·詹姆斯·普劳格（Phillip James Plauger）本人实现，因此在其头文件中也包含有普劳格本人的版权声明。普劳格是标准 C 中 stdio 库的早期实现者，由于其个人与 Microsoft 保持有良好的关系，所以 Plauger STL 是被用于 Microsoft Visual C++中的 STL 版本。该版本的源代码可以在 Visual C++的 Include 子目录下找到。不过因为这个 STL 版本不是开源的，所以这些源代码是不能被修改和销售的。此外，

需要提醒读者的是，其源码的可读性较差，如果是初学者建议还是不要看这个版本的 STL 代码了。

3．Rouge Wave STL

Rouge Wave STL 是由 Rouge Wave 公司实现的，它也是 HP STL 的一个继承版本。该版本被 Borland C++ Builder 所采用，读者可以在 C++ Builder 的 Include 子目录下找到所有头文件。不过，Rouge Wave STL 不是开源的，此外，它的性能也不是很好。尽管其源代码的可读性较好, 但是, 由于该版本已有一段时间没有更新且不完全符合标准，因此在 Borland C++ Builder 6.0 中，它已经被另一个 STL 的实现版本所取代。Borland C++ Builder 6.0 只是出于兼容性的考虑才保留了 Rouge Wave STL。

4．STLport

STLport 最初源于俄国人鲍里斯 · 米切夫（Boris Fomitchev）开发的一个项目，主要用于将 SGI STL 的基本代码移植到其他的主流编译器上。因为 SGI STL 是开源的，所以 STLport 才有权这样做。STLport 已经被 C/C++技术委员会接受成为工业标准，且在许多平台上都被支持。根据测试 STLport 的效率比 VC 中的 STL 要快。比 Rouge Wave STL 更符合标准，也更容易移植。Borland C++ Builder 已经在其 6.0 版中加入了对 STLport 的支持，并以此替换原有的 Rouge Wave STL。

5．SGI STL

SGI STL 是由 Silicon Graphics Computer System 公司实现的，其设计者和编写者包括斯特潘诺夫和麦特 · 奥斯丁（Matt Austern），同样它也是 HP STL 的一个继承版本。它是一个开源的 STL。SGI STL 被 GCC 所采用，因此可以在 GCC 的 Include 子目录下找到所有头文件。由于 GCC 对 C++语言标准的支持很好，SGI STL 在 Linux 平台上的性能相当出色。其源代码的可读性也很好。有兴趣的读者可以从 SGI 的网站上得到更详细的信息，并可以免费下载其源代码。对于希望深入理解 STL 的读者而言，参考 SGI STL 无疑是最好的选择。

本章参考文献

[1] 查尔斯 · 佩措尔德. 左飞，等译. 编码：隐匿在计算机软硬件背后的语言. 北京：电子工业出版社，2012.

[2] 乔恩 · 本特利. 黄倩，等译. 编程珠玑（第 2 版）. 北京：人民邮电出版社，2008.

[3] Lee Badger. Lazzarini's Lucky Approximation of π. Mathematics Magazine, No. 67: 83~91, 1994.

第 2 章

指针与数组——也谈中国古代兵制

本章是通向数据结构领域的入口。数组是最简单的、最基本的线性数据结构，它可以被认为是程序设计语言提供的一种已经封装好了的数据结构。此外，在本书后面，许多数据结构的实现也是以数组为基础的（例如顺序栈以及循环队列）。本书中用到的另外一种数据结构实现方法则以链表为基础（例如链式栈以及链式队列）。这个过程涉及了关于动态内存管理以及指针方面的内容，而且指针和数组之间还具有非常紧密的联系。因此，切实掌握本章所介绍的内容对于后面学习的开展将具有非常重要的意义。

2.1 指针

很多程序开发语言都不支持“指针”，例如 Java、Basic 等，尤其在面向对象思想中，指针被认为是应当禁止的，它显然与面向对象世界格格不入！然而尽管如此，C++还是从 C 语言那里将指针继承了过来，这也是 C++受到最多批评的地方。指针提供了对内存进行直接操作的手段，这使得 C/C++语言灵活而强大，但是在某种程度上也增大了风险，内存错误往往导致严重的后果却难以察觉。本节就介绍有关指针的内容，关于内存管理方面的问题将在本章最后讨论。

2.1.1 内存与地址

众所周知，现代计算机科学有两位最重要的奠基人，一位是英国科学家艾兰·图灵（Alan Turing），另一位是美籍匈牙利科学家冯·诺依曼（John von Neumann）。图灵的贡献是建立了图灵机的理论模型，奠定了人工智能的基础。而冯·诺依曼则是首先提出了计算机体系结构的设想，即所谓的冯·诺伊曼结构。冯·诺伊曼结构的一个核心理论要点就是将程序指令存储器和数据存储器合并在一起，即所谓的“程序存储”。依据这一原理，程序的数据和指令都存在内存中。从物理意义上讲，内存就是一块具有存储功能的半导体材料。任何需要被计算机执行的程序（包括指令和数据），都要先从外部存储器（包括硬盘、光盘等）调入内存后才能被执行。内存中同时持有很多数据，当前需要哪个数据，CPU 都会从内存中精确地找到它然后再让其参与运算。

CPU 是如何从内存中找到其想要的数据的呢？想象现实中的仓库，其实就可以找到答案。如果有一个很大的仓库，为了让所存储的物品容易管理，这个仓库通常会被划分为若干个房间，而且每个房间都有一个编号，根据这个编号就可以找到相应的房间，

然后找到相应的货物。

内存采取了同样的管理方法，首先内存被划分为许许多多个很小的存储单元，然后这些存储单元被编上了号，由于内存中的存储单元太多，所以这些编号往往是一个非常大的整数，例如 0x0019F878（这是一个十六进制数表示的大整数）。这些用来标识内存中各存储单元的大整数编号就是“ 地址 ”。换句话说，CPU 执行的机器码通过各内存位置的唯一编号来操作这些内存中的数据，这些编号就是所谓的内存地址，或简称地址。而编译器的工作过程就是将程序中的变量和操作翻译成对内存地址进行操作的机器语言。所以，在最终的机器码里其实是不存在变量名或变量类型的。

在更多的时候程序员都不太关心他们所定义的变量到底存储在内存的哪个位置，因为系统会为他们解决这些问题。他们只需要定义一个容易理解和记忆的“ 名称 ”来标识数据就足够了。但是 C++中提供了指针这个用以直接访问内存的手段。在说明指针的概念和用法之前先来看看计算机中的数据是如何在内存中组织的。在 Visual Studio 2013 环境下编写一段 C++程序，并在设置了合适的断点之后，进入调试模式。

```
#include "stdafx.h"
#pragma runtime_checks( "", off )

int _tmain(int argc, _TCHAR* argv[])
{
    int a = 97;
    int b = 98;
    int c = 99;
    return 0;
}
```

上面的程序定义了 3 个整型变量a、b和c，图 2-1 是该程序调试时的 Watch 窗口。到目前为止还没有介绍过&运算符，它的具体用法将在 2.1.2 节中介绍，现在只说明&a表示变量a在内存中的地址，同理，&b是变量b在内存中的地址，&c是变量c在内存中的地址。在第 1 章中就已说明过，C++中每个 int 型变量占用 4bit。所以a、b和c这 3 个变量的地址编号都相应地差 4，图 2-1 很直观地演示了这一点。

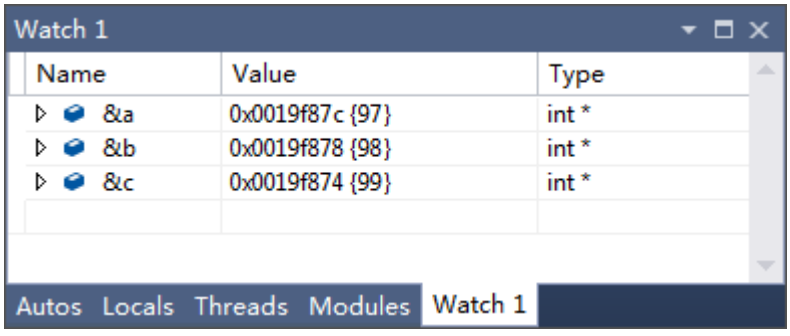


图 2-1　Watch 窗口

相信读者已经在脑海中有一个关于数据在内存的组织形式的大概模型了。简单地说，内存就是一段连续的存储空间，内存被均等地分为多个存储单元，每个单元都称为一个内存单元。数据在内存中按一定的顺序连续存放。为了便于管理，系统给每个内存

单元都编了一个号，称为内存地址。每个内存单元的地址都是一个很大的整数，例如0x0019F878。这就好比一家酒店有很多个房间，每个房间都有一个房号。当有顾客入住某个房间时，就相当于在内存中的某个存储单元放了一个数据。当有人来找酒店中的某个顾客时，他就按照房间号找到相应的房间，然后见到他想见的人。图 2-2 很形象地描述了内存中数据的基本组织形式。

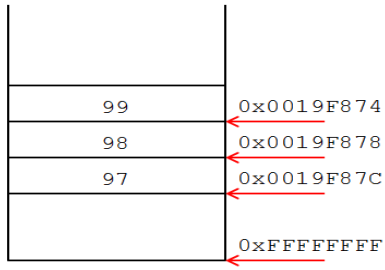


图 2-2　内存和数据

再来谈谈去酒店找人的例子。为了找到某人你需要到前台询问该人的房间号，然后再根据房间号找到该人，如图 2-3 所示。有没有一种更简单的方式呢？当然有。如果你已经知道房间号那就无须再向前台问询，可以直接去找人了。“直接找”当然比“间接找”要快得多。计算机中也提供了对于一个数据的“直接访问”和“间接访问”这两种访问方式。通过变量来访问数据就是间接的访问方式，通过地址来访问数据就是直接的访问方式。

图 2-3　到酒店找人

变量其实就是内存存储单元的一个“临时的别名”，而地址才是这个存储单元“永久的真名”，是不可以更改、也不能重复的。内存中不会有两个重名的内存存储单元，但是在程序中同一个变量名可能代表着不同的内存单元。只要变量的作用域不同，程序就允许变量同名。现实生活也是这样，两个人的名字可能相同，所以在这种情况下，每个人还有一个唯一编号来进行区分，在社会上这个编号可以是“身份证号”，在学校里这个编号可以是“学号”。内存中用来唯一标识和区别内存存储单元的就是“地址”。

计算机内部只能够通过地址来找寻内存单元。如果用一个别称（即变量）来标识存储

单元，计算机就需要先把这个别称翻译成地址，然后再去找相应的内存单元。因此这个过程是一个间接访问的过程。就像你去酒店找人，如果不知道房间号就要去前台问叫某个名字的人住在哪个房间。如果你已经知道存储单元的地址了，那么你就可以通过该地址直接去访问存储单元，而无须计算机底层对这个别名进行翻译。也就相当于你去酒店找人前已经知道那个人所在的房间号时就无须再向前台问询了。

假设 CPU 要处理一条指令，例如“ a + b ”，该指令包含两个数据a和b，CPU 需要知道a和b的具体内容是什么才能进行运算，于是它通过某些方法在内存中查找a对应的值，而a刚好就是一个内存地址，例如 0x0019F87C，然后 CPU 直接在内存中地址为 0x0019F87C 处将数据取出。这就是直接寻址。然后 CPU 又开始查询b的值，这时它发现b也是一个地址，但这个地址里并没有存放实际要被操作的数据，而是存放了另外一个地址，这个地址里存放的才是b真正对应的值，于是 CPU 才将值取出。这体现了一种最普遍的处理问题的思想——直接处理和间接处理。生活中这样的例子是非常多的，例如我们想在字典里查一个字的释意，第一种可能是已知这个字的释意具体所在的页数，那么就拿出字典直接翻到那页即可，另外一种是不知道这个字的释意在哪一页，但是我们可以在索引中查询，找到页码后再翻到那页。再比如有一把钥匙可以开一道门，但是为了安全这把钥匙并不是随身携带的，它可能被锁在某个保险柜里，这显然是一种间接访问。

下面终于可以引出指针和指针变量的概念了。在 C++中，一个变量的地址称为该变量的“ 指针 ”。指针也是一种数据，它当然也可以被存在一个内存单元中。如果定义一个变量专门用来存放另一个变量的地址，则它就是一个指针变量。即用来存储指针的变量，就称为指针变量。指针变量的值就是指针。如图 2-4 所示，pointer就是一个指针变量，这个变量里面存储着变量a的地址。

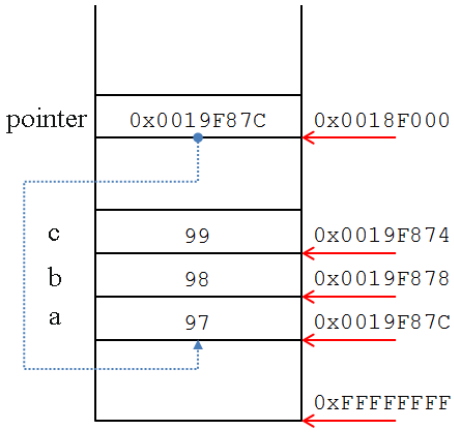


图 2-4　数据和指针

2.1.2　指针的语法

在 C++中，变量在使用之前必须被定义，明确其类型，并分配存储空间。指针变量必

须依托它所指向的变量，才能具有实际的意义。为了表示指针变量和它所指向的变量之间的关系，C 语言中使用“ * ”来表示这种指向关系。例如有如下定义

```
int i = 0;
int * p = i;
```

以上语句表示指针变量p指向了一个变量i，而* p就表示p所指向的变量，即* p就表示i。

一般的指针变量的定义形式为：

```
类型说明符 * 指针变量名;
```

如果希望改变指针所指向的内容，可以采用取地址符“ & ”，例如：

```
int i = 0;
int j = 0;
int *p = i;
p = &j;
```

上面这段代码首先定义了一个指针变量p并使它指向变量i，而后又使它更改指向变量j，此过程如图 2-5 所示。

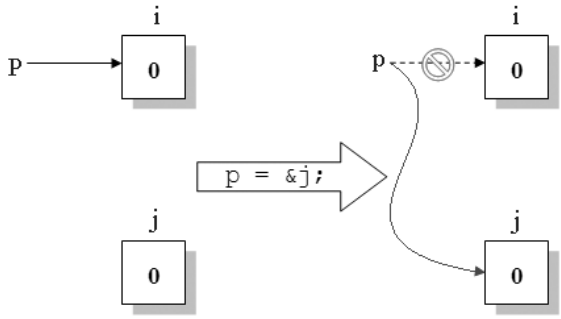


图 2-5　改变指针的指向

此外，int ＊p这个定义方式是需要我们深入理解的，其中int ＊可以作为一种类型名来看待，它表示p是一个int型指针，而* p则可以当成一个int型变量来使用。这种描述方式虽然便于读者理解，但也会常常引起误会。如果int ＊可以作为一种类型名来看待，那么下列定义语句将实现怎样的功能呢？

```
int * a, b, c;
```

相对于类型而言，可以认为*和变量名结合的关系更紧密些，所以上述定义的结果是a是一个指针变量，而b和c都是int型变量！如果想将a、b和c都定义成指针型变量则只能采取如下的方式。

```
int * a;
int * b;
int * c;
```

因为不同类型的变量在内存中所占据的空间大小是不同的，所以在定义指针时必须指定其类型，这对于指针的移位操作具有很重要的意义。如果定义变量i是一个整型变量，且初值为0，那么i++就是将i自加1。对于上例中的指针类型p，如果有操作p++，那么

对于指针变量p自加1意味着什么呢？因为p是一个int型指针，因此p++操作意味着将指针p移动4个字节（假设一个int型变量占据 4 字节的空间）。可见正确地定义指针类型是非常重要的。如果类型不匹配很可能得到错误的数据或发生访问越界，这些都是非常危险的行为。

此外，指针变量中存放的值只能是变量的地址，而不可以是其他的类型。将一个整数赋给指针变量也是错误的。尽管地址本身也是一个大整数，但是前面已经说过指针是变量的地址，这个地址的获得必须通过一个已经被分配了存储空间的变量来完成，变量的地址只能通过指针运算符*或取地址符&来获得，而不能直接指定一个具体的地址。因此，下面语句就是错误的。

```
int * a;
a = 0x5050;
```

再来考虑一下& * p和* &p的区别。因为*和&具有相同的优先级，按照从右向左的方向结合。对于& * p而言首先进行一次* p运算再取地址，就相当于&(* p) 或者&i，即变量i的地址。同理，* &p相当于* (&p)，就是先取地址再做*运算，* &p等价于p。

前面已经介绍过，指针提供了一种直接访问的方式。直接访问除了更加高效以外，它还能完成一些间接访问无法实现的功能，指针提供了更大的编程灵活性，这种灵活性甚至允许程序员透过层层屏障去操作计算机的底层。

2.1.3　使用指针变量

要想把指针运用得随心所欲，首先还是要搞明白指针和变量的关系。下面这段示例代码能够帮助读者理解一些令人困惑的问题。

```
#include "stdafx.h"
#include "iostream"
#pragma runtime_checks( "", off )

using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
    unsigned int a = 5;
    unsigned int *pint = NULL;

    cout << "&a = " << &a << endl << " a = " << a << endl;
    cout << " &pint = " << &pint << endl << " pint = " << pint << endl;
    cout << " &(*pint) = " << &(*pint) << endl << endl;

    pint = &a;
    cout << "&a = " << &a << endl << " a = " << a << endl;
    cout << " &pint = " << &pint << endl << " pint = " << pint << endl;
    cout << " &(*pint) = " << &(*pint) << endl << endl;

    *pint = 10;
    cout << "&a = " << &a << endl << " a = " << a << endl;
    cout << " &pint = " << &pint << endl << " pint = " << pint << endl;
    cout << " &(*pint) = " << &(*pint) << endl << endl;
```

求职面经点拨

请试回答附录题库中的第 4 题。该题综合考察了求职者对于 const 关键字和指针的理解。

```
    system("PAUSE");
    return 0;
}
```

编译并运行上述程序，其运行结果如图 2-6 所示。

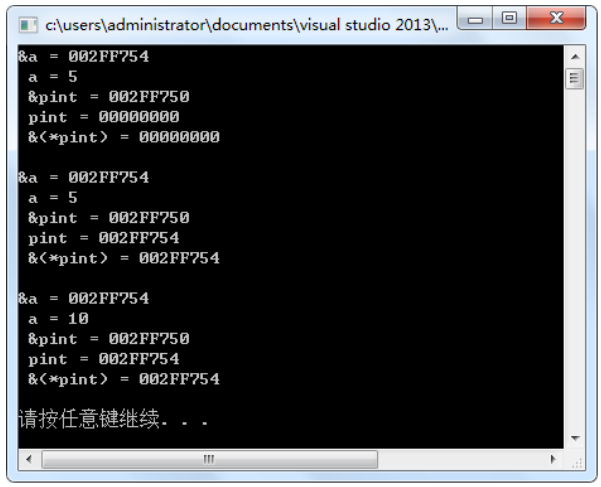


图 2-6　程序运行结果

我们将上述程序分成 3 段来研究。首先程序在最开始定义了一个整型变量a，并为其赋值5。此时，整型变量a的值是5，变量a所在内存单元的地址&a是0x002FF754。然后程序又定义了一个指向整型变量的指针pint，最开始该指针变量的值为 NULL，即它不指向任何变量。指针变量pint也被存于内存的某个存储单元中，它的地址&pint 为0x002FF750，刚好与&a相差4个字节。这是因为在我们的环境下整型变量占据4字节的内存空间，而这两个变量（一个整型变量和一个指针变量）在程序中的定义是相连的，因此它们在内存中的存储单元也是相邻的。如图 2-7 所示，整型变量a的值是5，因为它占据4字节，也就是32个二进制位（比特），所以用十六进制数来表示就是0x00000005，其地址&a = 0x002FF754。指针变量pint的值为0x00000000，这表示它不指向任何存储单元，指针变量pint的地址&pint是0x002FF750。

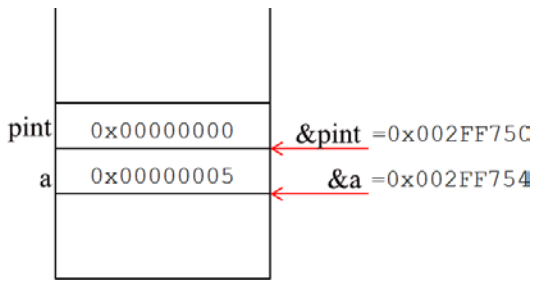


图 2-7　变量与地址

前面已经讲过，如果把int * pint这个定义方式中的int *作为一种类型名来看待的话，这个定义就表示pint是一个int型指针，而* pint可以当成一个int型变量来使用。而且在定义了一个指针变量p后，就可以通过* p来使用指针所指向的对象。从上述输出结

果中可以看到pint的值始终等于&(* pint)的值，如何理解表达式&(* pint)的意义呢？如果把* pint看作是一个整型变量，那么&(* pint)就是对此变量取地址，这个地址也就是指针变量pint所存储的值，因此pint始终等于&(* pint)。

程序继续执行，本来指针变量pint是不指向任何存储单元的，但语句“ pint = &a; ”将变量a的地址赋给了指针变量pint，于是整型指针pint也就指向了变量a。如图 2-8 所示，指针变量pint的值变成了0x002FF754。此时* pint则表示一个int型变量，该变量的地址等于pint的值。在本例中，pint的值就是整型变量a的地址，所以* pint就等同于整型变量a，所以它的值等于5。

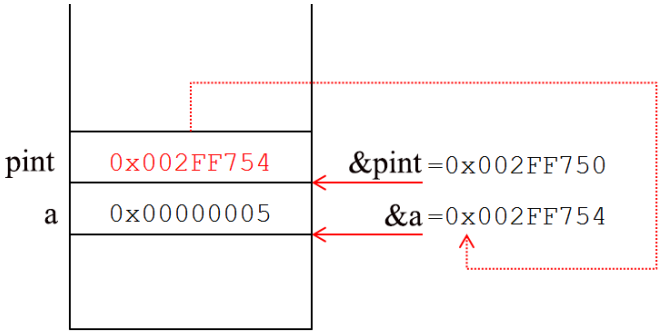


图 2-8　给指针变量赋值

在程序的最后一部分，由于* pint就等同于整型变量a，所以修改* pint也就相当于修改变量a。当执行了语句“ * pint = 10; ”后，变量a的值就由5变为了10。

上面这个程序非常详尽地演示了一个指针变量的使用方法。如果读者感觉这个程序中的每条语句自己都能非常明确地说出其作用，那么就表明读者已经基本掌握指针变量的使用方法了。

2.1.4　函数与参数传递

小问号

C++ 中函数参数传递的方式有哪 3 种?

C++中的函数分为两类，即普通函数和成员函数。无论是何种函数，它们的定义都具有相同的语法形式和组成部分。一般 C++中的函数定义主要分为 4 个部分：函数名、函数体、返回值类型和形参表(形式参数列表)。函数的使用者通过函数名来调用函数，函数体是函数功能的主体实现部分，返回值作为最后的处理结果被返回给调用者，形参表是调用者与函数之间进行参数传递的手段和媒介。一般函数定义的语法形式描述如下。

```
返回值类型 函数名(形参表)
{
    函数体;
}
```

每个函数有且仅能有一个返回值，返回值类型规定了该值的类型。特别地，返回值类型可以为空，用符号 void 标识。当函数返回值不为空时，函数体中必须含有 return 语句。若返回值为空，则可以省略 return 语句。

函数的形式参数列表内容一般为 type1 name1, type2 name2, …, typen, namen, 其中 type1, type2, …, typen 表示被传递参数的类型, name1, name2, …, namen 表示被传递的参数名。函数参数可以有多个，也可以没有，当无参数时形式参数列表可以用 void 标识，也可以省略不写。

C++中，函数的传递方式有 3 种：按值传递、指针传递和引用传递。

当使用按值传递方式时，函数内部将对参数做一个备份，函数对此备份进行操作。借由参数数值的备份来完成参数传递工作的传值方式称为按值传递。按值传递是 C++中使用的一种默认传值机制。这种机制的特点在于当被传递的参数值在函数体内发生改变时，调用主体中的原始数值不会发生改变。来看下面这个例子。

```
#include <iostream>

using namespace std;

void fun(int x)
{
    cout<< “x = ”<<x<<endl;
    x++;
    cout<< “x = ”<<x<<endl;
}

int main(int argc, char** argv) {

    int x = 0;
    cout<<"x = " <<x<<endl;
    fun(x);
    cout<<"x = " <<x<<endl;

    return 0;
}
```

运行结果如图 2-9 所示。因为函数 fun 采用了按值传递的参数传递方式，因此函数体内生成了参数x的一个备份，对这个备份值进行自加运算，于是这个备份的值由0变到了1。一旦函数 fun 返回，那么这个备份值的生命周期就结束了，返回主调函数后，原变量x的值并没有被改变，于是程序输出了0。

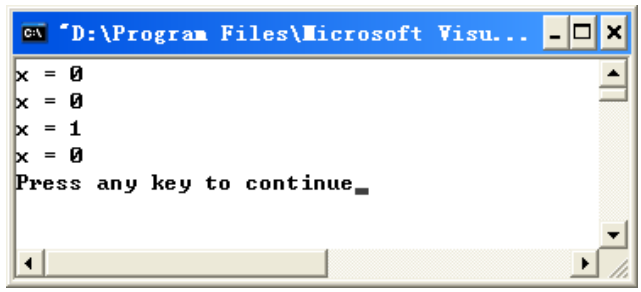


图 2-9　按值传递程序运行结果

C++也可通过按值传递的方法来传递一个对象。下面这个例子简单地定义了一个类，然后通过按值传递的方式将这个类的一个对象传递给了被调函数。

```
#include <iostream>

using namespace std;

class Student {

    string name;
    int age;

public:
    Student() : name(""), age(0) {}

    void set_age(int age) {this->age = age;}
    int get_age() { return age;}
    void set_name(string name) {this->name = name;}
    string get_name() { return name;}
};

void increment_age(Student s) {
    s.set_age(s.get_age() + 1);
}

int main(int argc, char** argv) {

    Student s;
    s.set_name("Wumingshi");
    s.set_age(20);

    increment_age(s);

    cout << s.get_age() << endl;

    return 0;
}
```

读者可以自己编译并运行一下上述程序，然后分析一下结果，看看是不是和预想的一样。显然通过按值传递的方式，年龄修改功能并没有成功执行。

无论是函数 fun 还是函数 increment_age 都采用了按值传递的方式，这无疑带来了一个问题，即它无法修改“ 外部变量 ”。对于函数 fun，主调函数中的变量x就是外部变量。实验证明改变局部变量的值不会对外部变量造成任何影响，这是因为它们被分别存储在不同的空间中。如果希望函数可以实现改变外部变量的功能，使用指针传递的方式不失为一个好的选择。既然已经知道按值传递仅仅是在函数体内部对参数的一个备份进行操作因而无法改变外部变量，那么就很容易想到只要获得所传参数的指针，然后直接对指针所指向的值进行修改就可以满足要求了。当进行函数参数传递时，不是传递一个普通变量，而是传递一个指针变量，这就是指针传递方式。来看下面的例子。

```
#include <iostream>

using namespace std;

void fun(int* p)
{
    cout<< "*p = "<<*p<<endl;
    (*p)++;
    cout<< "*p = "<<*p<<endl;
```

```
}

int main(int argc, char** argv){
    int x = 0;
    cout<<"x = " <<x<<endl;
    fun(&x);
    cout<<"x = " <<x<<endl;
    return 0;
}
```

编译并运行上述程序，运行结果如图 2-10 所示。可见外部变量的值被成功地修改了。

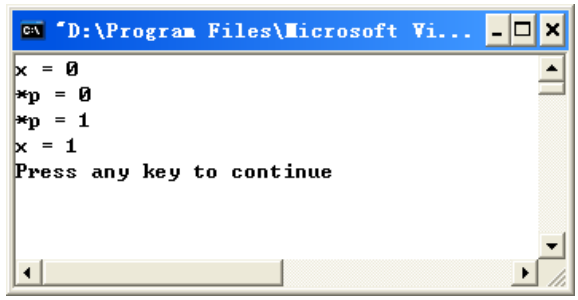


图 2-10　指针传递程序运行结果

C++也支持按引用传递参数的机制。C++中的引用（reference）可以理解为变量的一个别名，它是 C++中一种新的变量类型，是对 C 的重要扩充。当建立引用时，程序用另一个变量或对象的名字初始化它。此时，引用作为目标对象的别名使用，所以对引用的改动实际就是对目标的改动。为了建立引用，在编程时应先写上目标的类型，然后再写引用运算符“ & ”，最后是引用的名字。引用的命名规则与其他变量无异，且引用可以在该变量出现的任何地方出现，这称为独立引用。

尽管引用运算符与地址符使用的符号相同，且它们显然是彼此相关的，但绝不可将它们混为一谈。引用运算符只在声明时使用，它放在类型名后面。任何其他的“ & ”的使用都是地址操作符。

引用不是值，因而不占内存空间，声明引用时，目标的存储状态不会改变。前面已经讲过声明和定义的区别，既然“ 定义 ”的概念有具体分配空间的含义，那么引用只有声明，没有定义。

小问号

引用与指针有哪些不同？

引用的声明有些像指针，但必须在声明时就初始化，指出它引用的内存对象，否则会产生编译错误。有人觉得指针和引用非常相像，那么它们到底有哪些区别呢？二者的区别主要有 3 点。

首先，在任何情况下都不能使用指向空值的引用。也就是说存在空指针，但不存在空引用，一个引用必须总是指向某个对象，这个对象是一开始就指定了的。不存在指向空值的引用这个事实意味着使用引用的代码效率比使用指针要高。

其次，二者存在“ 合法性区别 ”，即在使用引用之前不需要测试它的合法性。相反，指针则应该总是被测试，防止其为空。这也是因为引用在声明时就已指定对象，且会一

直指向对象因此无须做合法性测试。

最后，指针可以被重新赋值以指向另一个不同的对象，这一点已经讲过。但引用则总是指向在初始化时被指定的对象，以后不能改变，当然，指定对象的内容是可以改变的。通过这点，可以明确引用一旦初始化，它就维系在一定的目标上，再也分不开。任何对该引用的赋值，都是对引用所维系的目标的赋值，而不是将引用维系到另一个目标上。

此外，在使用引用时还需要注意：首先，不能建立引用的数组。因为数组是某个数据类型元素的集合，数组名表示该元素集合空间的起始地址，它自己不是一个名副其实的数据类型。其次，引用本身不是一种数据类型，定义引用在概念上不产生内存空间，因此，没有引用的引用，也没有引用的指针。

引用提供了在 C++中传递变量地址的另外一种方法，引用传递和指针传递的作用是一样的，唯一的不同仅在于使用引用可以提高语法上的清晰性和易读性。

求职面经点拨

在涉及 C++的面试过程中引用常常是考察重点。请试回答附录题库中的第 5 题至第 9 题。

与按值传递不同，引用传递不再生成参数的一个数值副本，取而代之的是被调函数收到了一个引用，这个引用指向了主调函数所提供的真实参数。由此，引用传递也被用于构造一个能够改变外部变量的函数。即使在不需要改变外部变量的情况下，引用传递有时也可以被用来避免按值传递的低效性。来看下面这个使用引用传递方式进行传参的例子，这里再次修改了上面的程序。

```
#include <iostream>

using namespace std;

void fun(int& r)
{
    cout<< "r = "<<r<<endl;
    r++;
    cout<< "r = "<<r<<endl;
}

int main(int argc, char** argv){
    int x = 0;
    cout<<"x = " <<x<<endl;
    fun(x);
    cout<<"x = " <<x<<endl;
    return 0;
}
```

这个程序与按值传递的那段例子程序非常相近，唯一的不同仅仅是程序的第 5 行处，参数 r 被按照引用的方式来声明，易见其语法形式为 int& r。请不要和取地址符&相混淆，在此，符号&的作用只是告诉编译器参数的传递方式是引用传递。编译并运行程序，分析运行结果可以发现外部变量被成功地修改了。

引用传递也被用作一种传递大对象的机制。特别当对象非常大的时候，采用按值传递的方式会导致大量的时间消耗，因为对象的复制过程耗时严重。而采用引用传递方式则高效得多，因为它不需要进行复制工作。正如前面所说的那样，即使无须改变函数

的某个参数，引用传递作为一种高效的传参方式也不失为一个好的选择。这种高效性将在参数是一个大对象的情况下表现得尤为明显。在使用引用进行参数传递的时候，需要注意的是最好把被传递的引用声明为一个常量，即 const 类型，例如：

```
void pass_by_referene(const Type& data){......}
```

这是一种非常好的编程习惯。因为它将有效保护程序免受意外修改所导致的诸多问题的困扰，因为它保留了按值传递方式的安全性。

此外，采用引用传递方式来传递一个指针也非常有用。尤其当函数需要改变指针所存储的内存地址时，即当被传递的指针变量需要进行重定向时可以采用引用方式传递它。来看下面一个例子。

```
#include <iostream>

using namespace std;

void first_bigger(int*& p, int threshold) {

    while (*p <= threshold) {
        p++;
    }
}

int main(int argc, char** argv) {

    int numbers[] = {0, 12, 32, 44, 33, 5, 85, 45, 100, 75};
    int* result = &numbers[0];

    cout <<"Begin at: "<< *result << endl;
    first_bigger(result, 60);
    cout <<"Result is: "<< *result << endl;

    return 0;
}
```

该段示例程序的作用是从一组数字中找出第 1 个大于参考值的数字。指向数组的指针将在 2.2 节中介绍，到时候读者可以回来再看这个例子。这里只说明此例子传递了一个指针的引用，并在被调函数里修改了指针中存放的地址。这个操作简便而易懂，很好地实现了预期的功能。

2.2 数组

在计算机编程语言中，数组是相同数据类型的元素按一定顺序排列的集合，也就是把有限个类型相同的变量用一个名字命名，然后用编号区分它们的变量的集合，这个名字称为数组名，编号称为下标。组成数组的各个变量称为数组的分量，也称为数组的元素。本节将集中讨论 C++中数组使用方面的问题。

2.2.1　结构型数据类型

在程序设计语言中，每一个数据都属于某种数据类型。类型明显或隐含地规定了数据的取值范围、存储方式以及允许进行的运算。这些已经事先定义好的数据类型就是所谓的原子型。然而，当今世界纷繁复杂，人们每天面对着林林总总的事物，信息从各个方面以多种多样的形态呈现在人们面前，仅仅使用原子型数据很难有效地抽象现实问题。

因为人们所面对的数据量有时是非常巨大的，所以如果没有一个有效的方法来存储、组织和表示信息，那么信息将变得没有任何价值。即使面对一组相对较小的信息时，一个结构化的数据表示方法也是必不可少的。假想公共汽车站等车的人不按顺序争先恐后地抢着上车那么场面将是多么混乱！相反，如果人们排着队井然有序地逐个上车情况就好得多。

将原子型数据按照一定的规则重组，就可以形成结构型数据，这种将原子型数据组织成结构型数据的过程中所使用的模型，就是数据结构。或者说，数据结构就是一个信息的结构化表示，是为了将原子型数据组织成结构化数据而套用的某种组织模型。

问题导入

想想数组和中国古代军队的兵制之间有哪些联系？

数组可以被认为是一种最简单、最朴素的数据结构。数组的产生源于现实世界中最频繁出现的一类问题解决方式。

- 据《周礼》记载，我国古代军队“五人为伍，五伍为两，五两为卒，五卒为旅，五旅为师，五师为军。”可见，从西周时起，军队就是按伍、两、卒、旅、师、军编制排列的。
- 后来，商鞅变法，建立秦国新军连保制：“五人一伍，头目称伍长；十人一什，头目称什长；五十人为一属，头目称属长；百人一闾，头目为闾长，俗称百夫长……”
- 到东汉时，军制又为：“五人一伍，有伍长；十人一什，有什长；五十人一队，有队长；百人一屯，有屯长；二百人一曲，有军侯；千人一部，有司马或校尉，校尉为正、司马为副。”这也就是后来“队伍”一词的出处。

中国古代这种军队编制方式会给我们什么启迪呢？这里面至少蕴含了一种信息——将同种类型数据进行有序组织以便于管理的思想。这就引出了数组的定义，数组是一种简单而常用的线性数据结构。相对于整型、浮点型和字符型数据，数组较它们而言，无疑是一个重大的飞跃。数组是最简单、最基本的结构型数据类型，也就是说数组是由原子数据按一定顺序关系存放在一起的一个数据集合，组成数组的对象称为该数组的元素。再看东汉的军队编制方式，可知“ 五人一伍 ”，那么也就是说这个数组的容量是5，数组中的对象是“ 人 ”。

2.2.2　数组定义与初始化

C++中提供了通过数组来访问一系列同种元素的基本支持。数组类型声明的一般形式

如下。数组名是一个常量，它代表着数组元素在内存中的起始地址，也就是数组中第1 个元素的地址。数组可以是一维数组，也可以是N维数组，最常用的是一维数组和二维数组。定义数组的维数是通过“ [常量表达式 1][常量表达式 2]…[常量表达式N] ”来完成的，其中N即表示数组的维数。

```
数据类型 数组名[常量表达式 1][常量表达式 2]……；
```

其中，“ 常量表达式n ”称为下标表达式，它必须是一个 unsigned int 类型的正整数。数组的下标限定了数组中元素的个数，也用来标识数组中的元素。例如，可以声明一个拥有10个整数的一维数组如下。

```
int a[10];
```

注意其中10表示数组a中有10个元素，下标从0开始计算，也就是说这10个元素分别可以表示为a[0], a[1], a[2], …, a[9]。并不存在a[10]这个元素。当对a[10]进行操作的时候将发生越界访问。

此外，数组的容量在定义时就必须固定，C++不存在动态定义大小的数组，例如下面的定义是错误的。

```
int num;
cin>>num>>endl;
int a[num];
```

下面来看看一维数组是如何进行初始化的。对数组进行初始化可以采用多种方式，首先可以在定义数组时对数组元素进行逐一赋值。例如：

```
int a[10];
for(int i =0; i<10; i++)
{
    a[i]=i;
}
```

求职面经点拨

请试回答附录题库中的第 10 题。

上面的赋值方式等同于下面这种方式：

```
int a[10]={0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
```

即将数组元素的初始值依次放在一对大括号内。

当然，上面的方式存在一个问题，那就是当数组容量较大时，逐一对数组元素进行赋值是不现实的。所以可以仅对其中一部分元素进行赋值，例如：

```
int a[10]={0, 1, 2, 3};
```

这里定义数组元素的个数为10, 但仅对其中前4项进行赋值, 那么其余各项将默认为0。

此外，在定义数组时，对全部元素进行赋值，也可以省略数组长度。例如，下面这段代码默认数组a的容量为10。

```
int a[]={0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
```

一维数组是一个线性表，数组中的元素顺序存放，它支持对结构中任意元素的访问。访问的方式就是通过数组的下标[ ]来实现的。数组中第 1 个元素的下标是0，第 2 个元素的下标是1，第 3 个元素的下标是2……数组中的每一个元素都相当于同类型的一个

变量，对于同类型变量允许的任何操作都可以在对应的数组元素上进行。需要注意的是数组元素的下标表达式可以是任意合法的算术表达式，但其结果必须为正整数。例如：

```
int a[2*3] = {0, 1, 2, 3, 4, 5};
```

下面给出了一个小例子来演示数组元素的使用，它的作用是输出数组中所有元素的和。

```
#include <iostream>

using namespace std;

int main(int argc, char* argv[]) {

    int a[10] = {1, 32, 65, 2, 100, 34, 33, 21, 10, 1};
    int sum = 0;

    for(int i = 0; i < 10; i++){
        sum = sum + a[i];
    }
    cout << "数组中元素的和为 " << sum<< endl;
    return 0;
}
```

在 C++中使用数组的一个潜在危险就是 C++并不提供对于数组边界的检查。事实上，如果有一个容量为10的数组，读者可以尝试着去访问第11个值，甚至第100个值。这样做有些时候程序并不会崩溃。这取决于第11个值或第100值的性质。看下面一个例子。

```
#include <iostream>

using namespace std;

int main(int argc, char* argv[]) {

    int a[5] = {1, 2, 3, 4, 5};
    int b[5] = {6, 7, 8, 9, 10};

   cout << b[6] << endl;
   return 0;
}
```

图 2-11 是程序的运行结果。

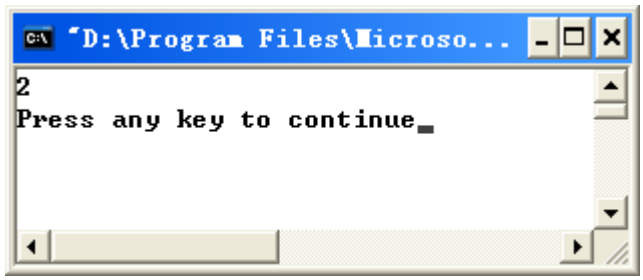


图 2-11　数组的越界访问

很明显，数组b中不存在b[6]这一项，但是由于b[6]这个位置存在一个合法值，于是程序仍然输出了结果（事实上b[6]的位置存放的是a[1]这个值）。尽管程序没有崩溃但至少发生了逻辑错误。如果非法位置上的值是比较敏感的，那个后果可能非常严重。在程序设计时务必杜绝这种情况的发生。

求职面经点拨

请试回答附录题库中的第 11 题至第 13 题。

另一种常用的数组是二维数组。二维数组相当于平面内的一个二维矩阵。根据前面的通用公式，可以很容易得到二维数组的定义语法，这里举例如下。

```
float array[2][3];//定义一个 2×3 的浮点型数组
```

注意array[2][3]表示的是2行3列，即第 1 个参数表示行数，第 2 个参数表示列数。同样可以通过下标来对数组中的任意元素进行操作。但是因为数组的下标仍然是从0计的，所以数组array中不存在array[2][3]、array[2][1]和array[0][3]等元素。

在定义了二维数组之后需要对它进行初始化。可以采用的方式首先是对数组中的元素逐一赋值，即将所有数据都写在一对大括号内。例如：

```
float array[2][3] = {1, 2, 3, 4, 5, 6};
```

对应的顺序是先顺序遍历第 1 行中的每个数据，再顺序遍历第 2 行中的每个数据……依此类推。但通常为了更直观、更易懂，还可以采取分行赋值的方式。例如：

```
float array[2][3] = {{1, 2, 3},{4, 5, 6}};
```

对于数组容量比较大的情况，可以选择其中部分元素进行赋值，例如：

```
flaot array[2][3] = {{1},{4}};
```

上面语句的作用只对每行首列进行了赋值，其他元素均被默认赋值为了 0，它对应的矩阵为：

$$\begin{bmatrix}1 & 0 & 0\\4 & 0 & 0\end{bmatrix}$$

当然也可以选择某行中的任意一列或多列来进行赋值，用这种方法对稀疏矩阵进行赋值特别方便。例如：

```
flaot array[2][3] = {{1},{0, 4}};
```

它对应的矩阵如下：

$$\begin{bmatrix}1 & 0 & 0\\0 & 4 & 0\end{bmatrix}$$

也可以跳过其中的某行，而对其他行进行赋值，例如：

```
flaot array[2][3] = {{},{0, 4}};
```

它对应的矩阵如下：

$$\begin{bmatrix}0 & 0 & 0\\0 & 4 & 0\end{bmatrix}$$

如果在初始化时对二维数组里的全部元素进行了赋值，则可以省略第 1 维的长度，但第 2 维的长度不能省略，例如：

```
float array[][3] = {1, 2, 3, 4, 5, 6};
```

为了进一步说明二维数组的使用方法，下面举一个简单的例子，下例实现的功能是对一个4 × 4数组进行转置。

算法

矩阵转置算法及其实现。在学习了 2.2.3 节之后，读者能否用指针对该算法的实现进行改写？

```
#include <iostream>

using namespace std;

int main(int argc, char** argv) {

    int a[4][4] = {{0, 1, 2, 3}, {4, 5, 6, 7},
                   {8, 9, 10, 11}, {12, 13, 14, 15}};
    int i = 0;
    int j = 0;
    int tmp = 0;

    for(i =0; i < 4; i++)
    {
        for(; j < 4; j++)
        {
            tmp = a[i][j];
            a[i][j] = a[j][i];
            a[j][i] = tmp;
        }
        j=i+1;
    }

    for(i = 0; i < 4; i++)
    {
        for(j = 0; j < 4; j++)
        {
            cout<<a[i][j]<<" ";
        }
        cout<<endl;
    }

    return 0;
}
```

通过二维数组的学习，可以很容易推广到高维数组的情况，这里不再赘述。

2.2.3　数组与指针

任何存储在内存中的变量都有地址，数组有地址，数组中的元素也有地址。2.2.2 节中我们介绍了关于指针的一些概念，那么如何使用指针来访问数组中的元素呢？本节将就此展开讨论。

2.2.2 节的学习告诉读者引用数组中的元素可以使用下标。除了使用下标，还有一种更经济、更高效的做法就是使用指针。首先可以定义一个指向数组的指针，例如：

```
int a[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
int * p;
p = &a[0];
```

这里首先定义了一个数组a，然后定义了一个int型指针变量p，并将数组a的首地址赋给了p。前面已经说过，数组中的元素是按顺序线性摆放的。于是通过对指针p进行操

作就可以轻松使用数组中的元素了。当然前面的代码也可以写成如下的形式。

```
int a[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
int * p = a;
```

数组a与指针p的对应关系如图 2-12 所示。

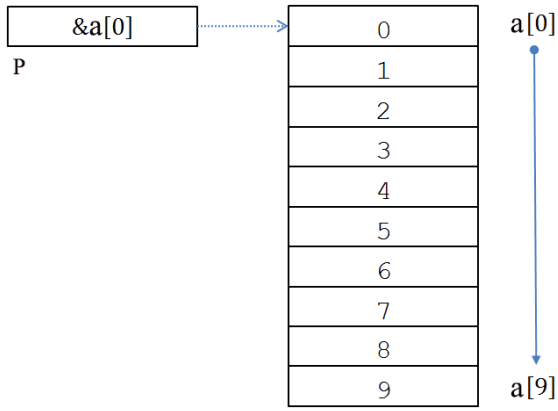


图 2-12　数组与指针的关系

下面有一个小例子可以用来说明二者的关系。

```
#include <iostream>

using namespace std;

int main(int argc, char* argv[]) {

    int a[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    int * p;
    p = &a[0];

    cout<< a[0] <<endl;
    cout<< &a[0]<<endl;
    cout<< &a<<endl;
    cout<< a <<endl;
    cout<< p <<endl;
    cout<< *p<<endl;

    return 0;
}
```

图 2-13 是程序的运行结果。可见指针p中存放着数组a的第 1 个元素a[0]处的地址，数组a的地址&a和数组名a的值都与首元素a[0]处的地址相同。＊p的内容就是首元素a[0]的值。

继续前面的例子，现在来考虑如何通过指针p来获得数组a中的其他元素。假设目前p中的值仍为&a[0]，基于这样的一个事实——数组元素在内存中是顺序、连续地存储的，即逻辑上相邻的数组元素在物理地址上也是相邻的，那么就可以通过将指针p向上或向下移动来访问数组a中的任意元素。所以，p + offset就是a[offset]的地址，或者说将指针p移动偏移量offset后，它就指向数组a中的第offset个元素。前面的例子告诉我们，

数组名a的值与首元素a[0]处的地址相同，所以，a + offset将具有同p + offset一样的意义。与之相对应的，*(p + offset)或*(a + offset)就是p + offset或a + offset所指向的数组元素a[offset]。另外，指向数组的指针变量可以带下标，所以p[offset]与*(p + offset)等价。

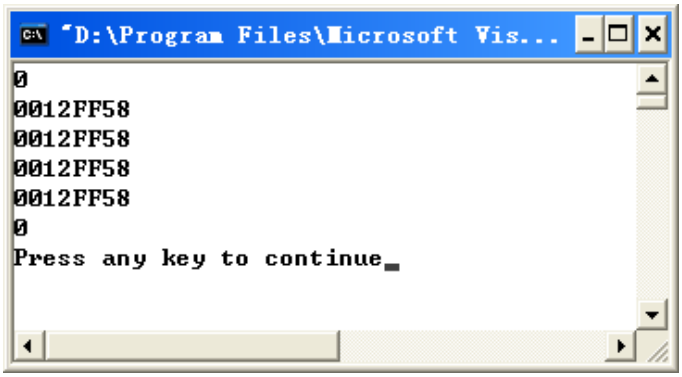


图 2-13　程序的运行结果

用指针访问数组元素是比较高效的。相对于传统的方法而言，它无须每次都重新计算地址，而使用下标符号[ ]，每次都需要进行变址寻址，这样的时间消耗在数组容量较大时是不可忽略的。特别当采用像p++这样的操作持续进行偏移访问时，效率的提高是很明显的。但是在运用指针变量进行运算时还是要非常小心。主要有如下几点比较容易出错。

- *p++、*(p++)与*(++p)的比较。

在 C++语言中，++运算和*运算具有相同的优先级，且运算结合的方向是自右向左，所以*p++和*(p++)的作用是完全相同的，即首先得到p所指向的内容，再将p向下移动一位（指向下一个元素）。但*p++与*(++p)的作用就不同了，*(++p)是先对指针p加1，再取*p。下面这个例子演示了三者的作用。

```
#include <iostream>
using namespace std;

int main(int argc, char* argv[]) {

    int a[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    int * p;

    p = &a[0];
    cout<< *(p++)<<endl;
    p = &a[0];
    cout<< *p++<<endl;
    p = &a[0];
    cout<< *(++p)<<endl;

    return 0;
}
```

程序的运行结果如图 2-14 所示。

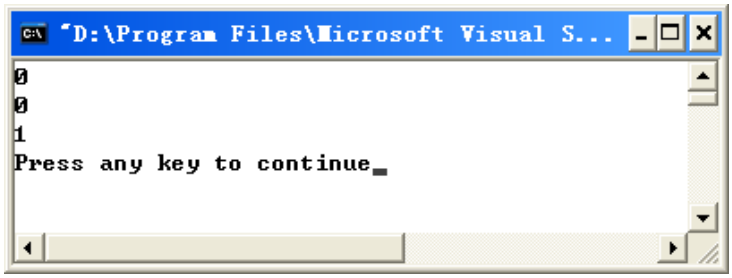


图 2-14　指针的运算

- (* p)++表示p所指向的内容加1，在上例中，若p = &a[0]且a[0] = 0，则(* p)++= 1。
- 如果p当前指向数组a中的第i个元素，那么有如下结论。
  - * (--p)与a[--i]的作用相同，p先自减，再做*运算；
  - * (p--)与a[i--]的作用相同，先进行* p运算，p再自减；
  - * (++p)与a[++i]的作用相同，p先自加，再做*运算。

需要说明的是，最后一点将指针运算和数组名a联系了起来。但两者并不能完全等同视之，即并非所有的运算都一样。特别地，可以通过指针运算来改变指针变量的值（也就是使指针向上或向下移动），但是不能将同样的操作运用于a，即不存在a++之类的运算。只是因为a表示的是数组首元素的地址，它是一个指针常量，所包含的值在程序运行时是不能被改变的。这相当于“ int i = 0; i++; ”是可行的，但“ int i = 1++; ”则是错误的。

使用指针访问数组时还需要注意的问题是，与数组存在的问题一样，指向数组的指针并不会做越界检查，这是非常危险的，不慎的访问很可能导致系统的崩溃因此要避免这种现象的发生。

前面的情况似乎都是在针对一维数组进行操作，那么如何使用指针对二维数组进行操作呢？事实上，二维数组可以看成是一个特殊的一维数组，而在内存中二维数组也是按行首尾相接线性排列的。这里定义一个二维数组a[3][3]，则这个二维数组中各项在内存中的排列为a00、a01、a02、a10、a11、a12、a20、a21、a22。可以试图将其推广到更加复杂和通用的情况下，例如，有一个二维数组b[M][N]，若将其转化成一个一维数组B[$M \times N$]，那么原二维数组中的b[m][n]项就对应一维数组中的B[$(m+1) \times (n+1) - 1$]。

基于上面的认识，可以编写一个简单的示例小程序如下。

```
#include <iostream>

using namespace std;

int main(int argc, char* argv[]){
    int a[3][3] = {0, 1, 2, 3, 4, 5, 6, 7, 8};
```

```
    int * p;
    p = &a[0][0];

    for(int i = 0; i<9; i++)
        cout<<*p++<<endl;
    return 0;
}
```

编译并运行程序，分析结果，读者可以看到二维数组被正确地遍历了。

2.2.4　数组的抽象数据类型

前面的介绍多偏重于数组的语法形式，本节中将把数组当成一种结构来看待。通过前面的学习读者应当可以理解一维数组其实是一个连续存储的线性表，而由于高维数组都可以转化成对应的一维数组，所以从这个角度来说，数组就是一个连续存储的线性表。

抽象数据类型是对一个数学模型及定义其上的运算的一种称谓。在程序设计语言中，抽象数据类型就是基于原子型数据类型构建的结构型数据类型以及定义在其上的操作。

对一个抽象数据类型进行定义时，必须给出它的名字及各种操作的名称，即函数名，并且规定这些函数的参数性质。一旦定义了一个抽象数据类型及具体实现，就如同生成了一种新的数据类型一样可以在程序设计中方便地使用它了。

用面向对象思想来描述抽象数据类型非常便利且易于理解。因为抽象数据类型的描述包括抽象数据类型的名称、数据、数据之间的关系和操作的集合等。

根据这些描述给出操作的具体实现后，使用者依据这些描述来使用抽象数据类型。用面向对象思想描述时，将数据抽象成属性，将操作抽象成函数，如此即可方便地建立抽象数据类型。

抽象数据类型的作用是为了方便描述现实世界。因此，使用者大可以只关心它的逻辑特征，而不需要了解它的存储方式。例如，数组这样的线性表，它的抽象数据类型可以抽象成如下数学模型：数据元素的集合，且集合内的元素所具有的关系是除第一个和最后一个外，每个元素有唯一的前驱和唯一的后继。对于这个集合中的数据可以进行的操作包括：查找或修改其中一个元素等。

数据结构、数据类型和抽象数据类型，这 3 个术语在字面上非常接近，但它们在含义上却存在区别。

数据结构是第 1 章中曾经介绍过的概念。它用来反映一个数据的内部构成，即一个数据由哪些成分数据构成，以什么方式构成，呈什么结构。数据结构有逻辑上的数据结构和物理上的数据结构之分。逻辑上的数据结构反映成分数据之间的逻辑关系，物理上的数据结构反映成分数据在计算机内的存储安排。

数据具有不同的类型，就像动物包括爬行动物、哺乳动物等，具有相同数据结构的数据属同一类，同一类数据的全体称为一个数据类型。在高级程序语言中，数据类型用

来说明一个数据在数据分类中的归属，它是数据的一种属性。这个属性限定了该数据的变化范围和表征含义等一些特征。

在高级程序设计语言中由语言本身提供的基本数据类型，也称原子型数据类型，对应于简单的数据结构；此外，高级程序设计语言也提供用户自定义数据类型的存在，使用者可以通过结构体等形式用原子型数据类型来构造新的数据类型，这称为结构型数据类型，它对应于复杂的数据结构；在复杂的数据结构里，允许成分数据本身具有复杂的数据结构，因此结构型数据类型允许复合嵌套。

最常用的数据结构是数组结构，数组结构的特点是：成分数据的个数固定，它们之间的逻辑关系由数组下标来体现。这些成分数据按照序号的先后顺序连续地排列起来。每一个成分数据具有相同的结构，也就是属于同一个数据类型。所有的成分数据被依序安排在一片连续的存储单元中（这一点在前面重点介绍过）。所以，数组结构是一个线性的、均匀的、其成分数据可随机访问的结构。

抽象数据类型可理解为数据类型的进一步抽象。它把数据类型和数据类型上的运算捆在一起，进行封装。引入抽象数据类型的目的是把数据类型的表示和数据类型上运算的实现与这些数据类型和运算在程序中的引用隔开，使它们相互独立。对于抽象数据类型的描述，除了必须描述它的数据结构外，还必须描述定义在它上面的运算。抽象数据类型上定义的过程和函数以该抽象数据类型的数据所应具有的数据结构为基础。由于抽象数据类型中充分体现了封装和抽象的特性，因此用面向对象的思想对它进行描述特别恰当和方便。

数组的抽象数据类型描述如下。

```
const int DefaultSize = 100;

template <class Type>
template <class Type>
class Array
{
    Type *elements;                                          //数组存放空间
    int ArraySize;                                           //当前长度
public:
    Array(int Size=DefaultSize);                             //构造函数
    Array(const Array<Type>& x);                             //复制构造函数
    ~Array() {delete []elements;}                            //析构函数
    Array<Type> & operator = (const Array<Type> & rhs);  //数组复制
    Type& operator [] ( int i );                             //取元素值
    int Length () const { return ArraySize; }            //取数组长度
    void ReSize (int sz);                                    //扩充数组
};
```

数组的抽象数据类型中，类的私有成员变量包括指针型变量 elements 和整型变量 ArraySize，其中 elements 给出了数组的数据集合，ArraySize 表示数组中元素的个数，并通过公共成员函数 Resize()来对 ArraySize 进行修改以动态地改变数组容量。下面给出了数组部分成员函数的实现代码。

```
template <class Type>
Array<Type>& Array<Type >::operator= (const Array<Type >& rhs)
{
    int n = rhs. ArraySize; // 取 rhs 的数组大小
    if (ArraySize != n) {
        delete [] elements;            // 删除数组原有内存
        elements = new Type[n];        // 重新分配 n 个元素的内存
        if (elements == NULL)          // 如果分配内存不成功，输出错误信息
        {
            ArraySize = 0;
            cerr << "存储分配错误！" << endl;
            exit(1);
        }
        ArraySize = n; //记录本对象的数组大小
   }
    // 从 rhs 向本对象复制元素
    Type* destptr = elements;
    Type* srcptr = rhs. elements;
    while (n--)
        *destptr++ = *srcptr++;
    return *this; // 返回当前对象的引用
}

template <class Type>
Array<Type> :: Array (int sz)
{
    if ( sz <= 0 ) {
        ArraySize = 0;
        cerr << "非法数组大小" << endl;
        return;
    }
    elements = new Type[sz];
    if (elements == NULL) {
        ArraySize = 0;
        cerr << "存储分配错误！" << endl;
        exit(1);
    }
    ArraySize = sz;
}

template <class Type>
Array<Type> :: Array (const Array<Type> & x)
{
    int n = x.ArraySize;
    ArraySize = n;
    elements = new Type[n];
    if ( elements == NULL ) {
        ArraySize = 0;
        cerr << "存储分配错" << endl;
        exit(1);
    }
    Type *srcptr = x.elements;
    Type *destptr = elements;
    while ( n-- ) * destptr++ = * srcptr++;
}

template <class Type>
Type& Array<Type> :: operator [] (int i){
```

```
    if ( i < 0 || i > ArraySize-1 ) {
        cerr << "数组下标超界" << endl;
        exit(1);
    }
    return elements[i];
}

template <class Type>
void Array<Type> :: ReSize (int sz) {
    if (sz >= 0 && sz != ArraySize) {
        Type * newarray = new Type[sz]; //创建新数组
        if (newarray == NULL) {
            cerr << "存储分配错" << endl;
            return;
        }
        int n = (sz <= ArraySize) ? sz : ArraySize; //按新的大小确定传送元素
                                                    //个数
        Type *srcptr = elements;  //源数组指针
        Type *destptr = newarray; //目标数组指针
        while (n--) * destptr++ = * srcptr++;
        delete [] elements;
        elements = newarray;
        ArraySize = sz;
    }
}
```

本书的在线支持资源中另提供有上述类的测试程序，限于篇幅这里不再详细列出，读者可自行下载并用于检验上述类的使用。

2.3 数组应用举例

本节将提供几个综合运用数组（以及指针）的实例，希望读者可以借由这些实际问题强化自身对于前面所学内容的掌握。

2.3.1 Z 字形编排问题

经典问题求解

Z 字形编排问题。Z 字形编排在数字图像编解码方面十分常用。

JPEG（Joint Photographic Experts Group）是一种常见的图像文件格式，也是目前静态图像中压缩比最高的一种图像文件格式，它综合运用了多种压缩技术而达到一种极高的压缩比例。JPEG 也是一个国际数字图像压缩标准，压缩技术十分先进，它用有损压缩方式去除冗余的图像和彩色数据，获取极高的压缩率的同时能展现十分丰富生动的图像。目前，它已被广泛地应用于多媒体和网络程序中。通常，在 JEPG 编码过程中，有一个非常重要的步骤，即 Z 字形编排过程。Z 字形编排过程大致是这样的：经过前期处理的图像被分为若干个8×8的小图像块，此时就从小图像块的左上角开始沿 Z 字形对图像元素进行遍历，并将遍历所得的结果重新写入等大小的图像块中，整个过程如图 2-15 所示。

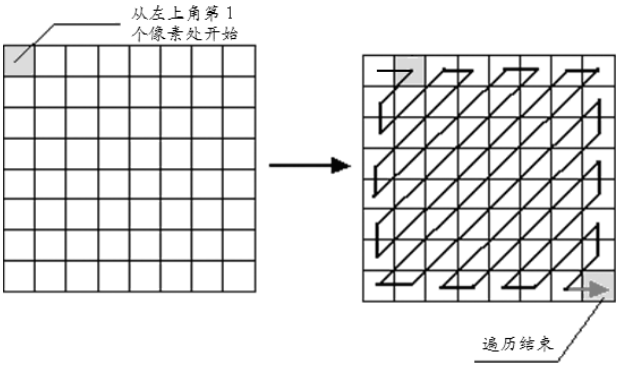


图 2-15　Z 字形编排方案

经 Z 字形排列后，原图像矩阵中的序号如图 2-16 所示。

$$\begin{bmatrix} 0 & 1 & 5 & 6 & 14 & 15 & 27 & 28 \\ 2 & 4 & 7 & 13 & 16 & 26 & 29 & 42 \\ 3 & 8 & 12 & 17 & 25 & 30 & 41 & 43 \\ 9 & 11 & 18 & 24 & 31 & 40 & 44 & 53 \\ 10 & 19 & 23 & 32 & 39 & 45 & 52 & 54 \\ 20 & 22 & 33 & 38 & 46 & 51 & 55 & 60 \\ 21 & 34 & 37 & 47 & 50 & 56 & 59 & 61 \\ 35 & 36 & 48 & 49 & 57 & 58 & 62 & 63 \end{bmatrix}$$

图 2-16　Z 字形排列后的图像矩阵示意

要实现这样一个 Z 字形排列，可能读者乍一看会感觉无从下手，但是在分析了 Z 字形遍历原矩阵过程中的走向规律，其实可以设计一个非常简单的算法来实现这种编排。对于原始矩阵matrix中的任意元素matrix[i][j]的遍历走向规律可以分为如下 3 种情况。

- 如果二维数组中的元素matrix[i][j]中纵坐标j是偶数，且$i=0$或者$i=7$，那么遍历路径在矩阵中的走向就是水平向右移动一格。
- 如果二维数组中的元素matrix[i][j]中纵坐标i是奇数，且$j=0$或者$j=7$那么遍历路径在矩阵中的走向就是垂直向下移动一格。
- 除上述规则以外的情况，如果二维数组中的元素matrix[i][j]的横纵坐标和$i+j$是偶数，则遍历路径在矩阵中的走向就是右上角移动一格；否则，若$i+j$是奇数，则遍历路径在矩阵中的走向就是左下角移动一格。

基于上述原则，完成矩阵 Z 字形编排功能的代码清单如下。

```
#include <iostream>
#include <iomanip>

using namespace std;

#define SIZE 8
```

```
int main(int argc, char** argv) {
    int matrix[SIZE][SIZE] = {0};
    int a[SIZE][SIZE] = {0};

    int i, j, x, y, value = 0;

    int * p;
    p = &matrix[0][0];
    //初始化矩阵
    for(i = 0; i<SIZE * SIZE; i++)
        *p++ = i;

    //打印原始矩阵
    cout<<"原始矩阵如下:"<<endl;
    for(i = 0; i < SIZE; i++)
    {
        for(j = 0; j < SIZE; j++)
            cout<<setw(4)<< *(*(matrix + i) + j);
        cout << endl;
    }

    i = 0; j =0;
    //进行Z字形编排
    for(x = 0; x < SIZE; x++)
        for(y = 0; y < SIZE; y++)
        {
            *(*(a + i) + j) = *(*(matrix + x) + y);

            if((i == SIZE-1 || i == 0) && j%2 == 0)
            {
                j++;
                continue;
            }

            if((j == 0 || j == SIZE-1) && i%2 == 1)
            {
                i++;
                continue;
            }

            if((i+j)%2 == 0)
            {
                i--;
                j++;
            }
            else if((i+j)%2 == 1)
            {
                i++;
                j--;
            }
        }

    cout<<endl<<"经过Z字形编排后的矩阵如下:"<<endl;
    for(i = 0; i < SIZE; i++)
    {
        for(j = 0; j < SIZE; j++)
            cout<<setw(4)<< *(*(a + i) + j);
        cout<<endl;
```

```
    }
    return 0;
}
```

请读者完成编码后编译运行程序，并观察输出结果。此外，这个算法不仅对8 × 8的矩阵有效，对于 SIZE 取其他值的情况仍然有效，读者不妨试试看。

2.3.2　大整数乘法问题

经典问题求解

大整数乘法问题。算法实现中大整数的输入方向与正常的方向是相反的，读者能否尝试对其进行改写？

应用数组可以实现的算法和可以解决的问题非常多，这里举一个非常典型的例子——大整数乘法。下面这个例子所采用的方法和思想都非常巧妙。计算机的精度是有限的，因此单纯使用程序设计语言提供的原子型数据类型来完成两个大整数的乘法显然是不切实际的。可以考虑用两个数组来分别存储一些小于 10 的整数，这些数字按顺序排列在一起，分别表示一个大整数每一位上的数字，于是大整数的存储问题就解决了。然后按照基本乘法规则对这两个大整数进行运算即可。下面给出了程序的代码清单。

```
#include <iostream>
#include <memory.h>

#define SIZE 14

using namespace std;

// 返回位数为 size1+size2
int* multi(int* num1, int size1, int* num2, int size2)
{
    int size = size1 + size2;
    int* ret = new int[size];
    int i = 0;

    memset(ret, 0, sizeof(int) * size);

    for (i = 0; i < size2; ++i)    {
        int k = i;
        for (int j = 0; j < size1; ++j)
            ret[k++] += num2[i] * num1[j];
    }

    for (i = 0; i < size; ++i)     {
        if (ret[i] >= 10)
        {
            ret[i+1] += ret[i] / 10;
            ret[i] %= 10;
        }
    }
    return ret;
}

int main(int argc, char** argv) {

    int num1[SIZE] = {1,2,3,4,5,6,7,8,9,1,1,1,1,1};//第1个大整数 11111987654321
    int num2[SIZE] = {1,1,1,2,2,2,3,3,3,4,4,4,5,5};//第2个大整数 55444333222111

    int* ret = multi(num1, SIZE, num2, SIZE);

    for (int i = SIZE*2-1; i >= 0; i--)
```

```
        cout << ret[i];

    delete[] ret; //释放内存

    return 0;
}
```

完成编码后，编译并运行程序即可。

经典问题求解

九宫格问题。有了这个程序，你也要赛过黄蓉了！

2.3.3 九宫格问题

在武侠小说《射雕英雄传》中有这样一个桥段。隐士瑛姑终日里苦修奇门异术，不想却被一道算题所困，虽百思而不得其解。问题最终被天资聪颖的黄蓉所破，并留有口诀：戴九履一，左三右七，二四有肩，八六为足，五居中央。这道算题就是著名的九宫格问题。九宫格是一种古老的数学游戏，它要求在三纵三横的一个矩阵中，填入从1到 9 这 9 个数字，并且纵向、横向、斜向上的 3 个数字之和皆相等。类似的问题在数学上被称为幻方。九宫格用现在的语言来描述，就是一个3×3的幻方问题。

下面就来设法求解推广的奇数九宫，即5×5、7×7，……的幻方问题。为此，给出一个求解口诀：“一居上行正中央，依次斜填切莫忘；上出框时向下放，右出框时向左放；排重便在下格填，右上排重一个样。”程序接收一个用户输入的奇数，并以此作为幻方矩阵的大小，动态地创建二维数组。为了实现该功能，下面的示例程序将使用指向指针的指针来生成二维数组。

```
#include <iostream>
#include <iomanip>
#include <memory.h>

using namespace std;

int main(int argc, char** argv)
{
    cout<<"请输入幻方的大小 n（n 是一个大于 1 的奇数）：";
    int n=1;
    cin>>n;
    cout<<endl;
    int **a = new int*[n];
    for(int i=0; i<n; ++i){
        a[i]=new int[n];
        memset(a[i], 0, n*sizeof(int));
    }

    int row=0;
    int col=n/2;

    for(int i=1; i<=n*n; i++) {
        a[row][col]=i;
        row--;
        col++;

        if(row<0&&col>=n)
        {
            col--;
            row+=2;
```

```
        }
        else if(row<0)
        {
            row=n-1;
        }
        else if(col>=n)
        {
            col=0;
        }
        else if(a[row][col]!=0)
        {
            col--;
            row+=2;
        }
    }

    for(int i=0;i<n;i++)
    {
        for(int j=0;j<n;j++)
            cout<<setw(6)<<a[i][j];
        cout<<endl;
    }

    for(int i = n; i > 0;)
        delete[] a[--i];
    delete[] a;

    return 0;
}
```

请读者完成编码后，编译并执行程序。

2.4　动态内存管理

在解决生活中的很多问题时，实际接触处理对象的容量之前，并不容易精准地确定问题的规模。例如，一个培训学校要招收 300 名新生，并将他们分配到 3 个班级，即每个班级 100 人。可是这一年学生报名非常火爆，有超过 500 人报名参加培训。于是这个培训学校就临时增设了 2 个班级。然后，这个学校开始为这 500 名新生登记入学，在向计算机中录入新生的姓名时又出现了新问题。学生管理系统所支持的个人名字最长只能录入 4 个汉字，而今年的新生中有几个少数民族学生，名字至少都有七八个字……生活中这样的例子还有很多，比如一条公路上每天的车流量，一列火车上每天乘客的数量，一个百货公司中某一件商品每天的需求量等，这些问题的规模尽管可以根据经验做事前估计，但却都不能精确预测。对于类似的问题，在软件开发中，就需要使用动态内存管理的方法来解决。

小提示

C++ 中的动态内存管理在实际开发中具有非常重要的意义，IT 公司招聘的笔试与面试中也常有涉及。

2.4.1　关键词 new 和 delete

数组在定义时，系统会为其分配固定大小的内存空间，这被称为是静态内存分配。静态内存分配的好处在于实现简单，操作方便，但也致使程序的可伸缩性大大降低。因为在很多的情况下，程序员无法预先确定要使用多大的数组。所以这种内存分配方式

的缺陷也是显而易见的。正如前面所列举的那些例子一样，在大多数情况下定长意味着会浪费宝贵的内存空间，而在某些特殊情况下，当定义的数组不够大时，可能引起下标越界错误，甚至导致严重后果。

求职面经点拨

请试回答附录题库中的第 14 题和第 15 题。

为了解决这样的问题，就需要使用动态内存分配。所谓动态内存分配就是指在程序执行的过程中动态地分配或者回收存储空间的分配内存的方法。由于动态分配不需要预先分配存储空间，而且分配的空间还可以根据程序的需要扩大或缩小，因此它能够有效地克服静态内存分配所带来的种种弊病。

创建一个新的类对象时，通常需要完成的工作包括：首先，为对象分配内存；其次，调用构造函数来初始化那块内存。程序员应当保证初始化过程正确进行，这一点非常重要，因为初始化的问题是程序出错的主要原因之一。尽管 C++中通过调用默认构造函数的方式强制对象的初始化过程，但谨慎的程序员应当尽量避免使用默认构造函数，而养成自己写初始化函数的习惯。

无论是为普通变量开辟空间，还是为类对象开辟空间，都可能有多种可选的方式。关于内存的组织形式和程序在内存中的存储运行方式已超出本书的范围，这里不再赘言。在此仅简单地说明几种程序运行时用于存储变量的内存区域。

第 1 部分称为静态存储区域，这部分存储空间在程序开始之前就可以分配，且这个存储空间在程序的整个运行期间都存在。例如，全局的静态变量就存在部分内存中。

第 2 部分被称为栈，栈中包含了带有函数返回地址的活动记录，以及当前函数使用的本地变量。对于一个被调用的子函数来说，函数体是包含在一对花括号中的一系列代码。一旦函数执行完毕，也就是说程序执行出了右花括号时，栈空间的存储单元就会被自动释放。

第 3 部分被称为堆，堆是区别于栈、全局数据区和代码区的另一个内存区域。它允许程序在运行时（而非编译时），申请内存空间。堆的存在为程序设计提供了灵活性，例如递归的实现就需要语言提供对动态内存分配的支持，堆意味着可以在任何时候分配内存并自适应地决定具体需要多少内存。本节所说的动态内存管理指的都是对于堆上的操作。

请注意，这里说的“栈”和“堆”与本书后面介绍到的“栈”结构及“堆”结构并非是等同的概念。这里的“栈”和“堆”是程序运行时用于存储变量的内存区域类型，而非某种特定的数据结构，读者应加以区分。

在 C++中，程序员可以利用关键字 new 和 delete 来实现内存空间的动态管理，new 可以根据具体对象的不同而自动开辟合适的内存空间，而 delete 则负责回收由 new 开辟的内存空间，它们帮助程序在运行时从堆中分配存储单元。为普通的变量申请内存空间，可以使用下面的语法规则。

```
new int;                    //开辟一个存放整型变量的空间，返回一个指向整型数据的指针
new int(100);               //开辟一个存放整型变量的空间，并指定其初值为 100
float * p=new float(3.14);//开辟一个存放实数的空间，并指定该实数的初值为 3.14,
```

```
//将返回的指向实数数据的指针赋给指针变量 p。
```

下面简单介绍使用 new 创建类对象的语法，来看下面这段示例代码，该程序首先定义了一个类 Point, 然后主程序分别创建了该类的两个对象。“ Point * point1 = new Point; ”会调用默认构造函数来初始化类对象；而 “ Point * point2 = new Point(3, 5); ” 则调用重载构造函数来初始化类对象。最后用 delete 释放内存时会自动调用析构函数。

```
#include <iostream>

using namespace std;

class Point
{
    int x;
    int y;

public:

    Point()
    {
        x = 0;
        y = 0;
    };
    Point(int xx, int yy)
    {
        x = xx;
        y = yy;
    };
    ~Point(){cout<<"END"<<endl;};

    int getX(){return x;};
    int getY(){return y;};
    void setX(int newX){x = newX;};
    void setY(int newY){y = newY;};
};

int main(int argc, char** argv) {

    Point * point1 = new Point;
    Point * point2 = new Point(3, 5);

    cout<<point1->getX()<<endl;
    cout<<point2->getX()<<endl;

    delete point1;
    delete point2;

    return 0;
}
```

还需要说明的是，当 new 操作符被执行后，如果有足够的内存空间，则 new 按要求分配一块内存, 并返回指向该内存起始地址的指针 ;如果内存空间不够, 则返回 NULL。

在 C++里同样可以使用 new 来为一个数组分配内存空间，并相应地使用 delete 来将其释放。其具体语法如下例所示。

```
Point * pt = new Point[100];
```

上述代码使用 new 在堆上创建了一个含有100个对象的数组，并把返回的指针赋给了指针变量pt。这样就在堆上为100个 Point 对象分配了足够的内存并为每一个对象调用了构造函数。那如何将这个数组所占据的空间释放呢？如果写成：

```
delete pt;
```

这就表示把数组中的第1个对象释放了，也就仅仅只是为第1个对象调用了析构函数，而另外99个析构函数调用没有进行，所以这是错误的语句。正确的语句应当采用下面这样的语法来实现。

```
delete [] pt;
```

空的方括号告诉编译器产生从数组创建时存放的地方取回数组中对象数量的代码，并为数组的所有对象调用析构函数。

2.4.2 避免内存错误

求职面经点拨

请试回答附录题库中的第 16 题。

本节将要讨论的内容跟实际的程序编写联系非常紧密。C++中支持指针操作，一方面最大程度上提供了编程灵活性，但同时带来了很多安全隐患。内存管理上的错误是用C/C++语言进行程序设计时最可怕的一类错误，它的危险首先原自它的不易察觉性，另一方面不得不承认这种错误的结果也常常是非常致命的。通常的内存错误被归结为以下 4 点：内存泄漏、重复释放、坏指针问题和超量写内存。

1．内存泄漏

在分配了一块内存空间之后，如果不再需要这些数据就应当考虑将其释放。如果被分配的空间不再需要时，程序员没有将其释放，那么这块空间将随同程序运行而一直存在。这在极少数的特殊情况下也有可能是正常的，但更多的时候是非常危险的。现在举一个简单的例子来说明这种情况可能带来的影响。假设某人处在一个封闭的空间内，他手里有一定数量的氢气球，氢气球飘浮在空中。每隔一定的时间就有一根栓氢气球的线断掉，然后氢气球飞向更高的空间。于是这个人就再做一只新的氢气球。

如此继续下去会发生什么情况呢？空间中气球的数量越来越多，但却都无法收回，最终空间内必然被气球填得满满的。断了线的气球就像是藏在某处的宝藏，尽管人们都知道有这么个宝藏存在，但是没有人能够找到它，无法找到的宝藏就没有任何意义。对于一个需要运行较长时间的计算机程序而言，如果它只是一味地分配而不进行回收，那么最终系统资源无疑将被耗尽。于是程序在这个过程中将运行得越来越慢，最终系统陷入瘫痪。在计算机中有很多程序要运行很长时间，操作系统实际上就是一个“死循环”。还有很多网络服务程序也会运行很长时间。即使不像上述例子那样极端，也的确存在很多程序尽管不一定会运行特别长的时间，但是它对内存的每次操作都有可能消耗很大的空间。比如一个图像处理程序。它打开的图片文件可能有几百字节，也有可能是几百兆，谁知道呢？如果内存仅是分配而忘记了回收，或是没有得到很好的回收，那么内存资源就好像一个漏了的水箱中的水不断地泄漏，直到最终耗尽为止——这就是所谓的内存泄漏。因此，必须杜绝这种现象的发生。

内存的泄漏通常是由回收失败导致的，例如下面这个例子，被分配的内存在函数的最后一行进行了释放，但当执行到 “ if (End()) return; ” 处时，函数如果满足结束条件就会直接返回，于是发生了回收失败。异常、错误和其他各种 throw 和 catch 语句经常是导致内存泄漏的原因。

```
void Memory_Leak()
{
    int * list = new int[100];
    ... ...
    if (End()) return;
    ... ...
    delete[]  list;
}
```

还有一种可能引起内存泄漏的原因是忘记了释放一个数据结构的某些部分。例如定义以下结构。

```
typedef struct {
    char *name;
    int number;
    int age;
    char *address;
    int phone;
} Student;
```

对于 Student 这个结构有一个相关函数如下。

```
void Memory_Leak()
{
    Student* s;
    s = new Student;
    s->name = new char[100];
    ... ...
    s->address = new char[50];
    ... ...
    delete s;
}
```

上面的例子中，一个 Student 结构体被分配了内存并在程序结束的时候内存空间得以释放，但这个结构体的一些域并没有被释放，例如 name 和 address 都被分配了空间，但它们没有被释放。这里要注意结构体被释放并不表示它的域也被释放了！

2．重复释放

分配内存必须释放，而且每次分配仅能对应一次释放。如果释放一个根本不存在的指针，抑或对于一个指针重复释放都会引起不必要的麻烦。要是释放函数能够自动检查将要被释放的空间是否已经被释放过了该有多好，然而并不是所有的存储分配器都留用足够的空间来记录那些内存块的释放和分配状态。何况内存分配的效率对于整个程序的运行表现影响巨大，如果在运行时反复进行检查势必导致程序效率大幅降低。当然，如果这个释放函数并不是经常被调用，而且出于程序的安全性和可读性考虑可以试着写一个防止重复释放的小函数，如下定义了一个宏，它可以用来避免重复释放。

```
#define SAFE_DELETE(p)    \
{                         \
```

```
    if ((p) != NULL) \
    {                    \
        delete (p);      \
        (p) = NULL;      \
    }                    \
}
```

更多的情况下，如果可以人为地避免重复释放也就无须再费这些周折了。所以还是需要来介绍一下可能发生重复释放的情况，首先来看一个小例子。下面的例子对同一空间进行了重复释放。

```
void Twice_Free(x)
{
   ... ...
   delete[] x;
}

int main(int argc, char** argv)
{
    int *x = new int[10];
    ... ...
    Twice_Free(x);
    ... ...
    delete[] x;
    return 0;
}
```

来分析这个例子，程序最开始开辟了一块内存空间用来存储一些整型数据，并在最后将这些数据释放。但是需要在程序运行过程中调用一个函数“Twice_Free(x)；”，在这个函数中 x 被释放过了一次，因此程序发生了重复释放。这是重复释放最容易发生的情况。因为在很多情况下，指针可能发生了传递，而传递的仅仅是地址，也就是说在一个程序中可能存在多个指针同时指向一块内存空间，对于其中任何一个指针的释放都会对这块空间进行回收。但由于其他指针貌似并没有回收，于是粗心的人就会画蛇添足地进行重复释放。可以举一个更极端的例子来说明这种现象，如下面一段代码。

```
BYTE* temp_buffer1 = new BYTE[100];
... ...
BYTE* temp_buffer2 = temp_buffer1;
... ...
BYTE* temp_buffer3 = temp_buffer2;
... ...
delete [] temp_buffer1;
delete [] temp_buffer2;
delete [] temp_buffer3;
```

事实上，temp_buffer1、temp_buffer2 和 temp_buffer3 都指向同一块内存区域，无须释放 3 次，仅仅释放其中之一就可以达到目的。

程序访问一段已经被释放了的内存空间可能引起很多危险。因为一旦内存块被释放，它其中的数据就有可能被应用程序或堆分配管理器修改。恰恰在修改之后，这块被占据的内存再次被其他的数据块访问到，后果就不堪设想了。一方面你所读到的程序其实是无用的，应用了这些没有任何意义的数据之后整个系统的运作都有可能受到影响，更糟的情况是你对这个块进行了写操作，也就是说你改写了本不该属于你的数据，结

果程序就崩溃了！下面给出了一个小例子，在这个例子中程序引用了一个已经被释放了的指针。

```
void Twice_Free(x)
{
   ... ...
   delete[] x;
}

int main(int argc, char** argv) {
    int *x = new int[10];
    ... ...
    Twice_Free(x);
    ... ...
    Function(x); // 错误！
    ... ...
    return 0;
}
```

一种避免上述错误发生的方法是当一个指针被释放时，就随即将其赋值为 NULL，前面的宏定义里这样做了。但如果该指针同时存在多份副本的时候，仅对其中之一赋 NULL，并不能从根本上预防错误的发生。总之，务必保证被分配的内存块仅被释放一次，保证已经被释放了的指针不会再次使用。如果这个指针曾经被复制了，就必须保证当它所指向的内存区域被释放之后，所有的副本都不应当被使用。

3．坏指针问题

内存管理方面的错误最主要是由坏指针问题所导致的，坏指针是一类问题的总称，它可能是由很多原因引起的。如果一个指针没有按预期指向应当指向的位置，那么当废弃这个指针时，一个通常的错误就发生了。

在这种情况下，最乐观的结果是指针终止于一些无效的内存空间。当发生这种情况时，程序经常会被立即终止。而可能发生的最糟糕的情况是一些随机的内存位置被修改了，但程序却继续运行下去。最终程序没有一点征兆地崩溃了，导致程序崩溃的数据看似与问题毫不相关，因为程序不知道从何时起就开始使用错误的数据了！这样一来，要想查出问题到底出在哪可是相当不容易的。即使非常有经验的程序员也常常被这些稀奇古怪的数值搞得非常头疼而不知所措。如果程序运行到最后没有崩溃，也没有提示任何错误并输出了结果（当然是错误的结果），于是错误被隐藏了起来，这无疑给软件使用者埋下了一颗定时炸弹，最终所导致的经济损失很可能是无法估量的。

如果你曾经遇到这种问题并修正了它，或许你会把出错的地方记得更加牢靠。因为没有谁能够总结出所有可能发生的错误，因此读者只能从实践中学会如何避免犯这些错误。尽管没有秘诀来预知所有错误，但是这里笔者还是极力向读者传授一些经验，这些经验性的方法和技术可以引导读者少走弯路并尽快找到错误的所在。

坏指针是从哪来的呢?一个不可忽视的根源就是没有被很好初始化的数据。假设你声明了一个本地指针但忘记了初始化它，因为变量是被储存在栈里的，而栈可能充满了那些来自先前活动记录的数据，恰巧这些数据没有被丢弃，那么指针就可能带有它们的值。

由堆来分配的对象存在同样的初始化问题。C++能够通过初始化函数来帮助避免这些问题，当一个类的实例被创建后，初始化函数会被自动调用，无论是在堆上还是在栈上。写初始化函数是一个非常好的习惯，它的确非常有效。如果不知道正确的初始值，那么就将指针赋为 NULL，那样它将不会指向任何数据。大部分环境下，NULL 是一个无效的地址，因此任何尝试读取数据的行为都会被立即中止。于是这些错误将相对比较容易定位和修正。

4．超量写内存

有时尽管指针被正确地初始化了，但程序员却不正确地进行了指针运算或者错误的内存地址操作，这同样会引起内存管理上的错误。

第一个常见的错误发生在使用指向数组的指针，数组的越界访问就是一个很容易犯的错误。容量为array_size的数组a是不存在元素a[array_size]的。如果使用指针不慎访问了这个地址，就可能会改变程序里其他数据的值而导致程序的崩溃。

另一个可能的错误是分配了不足的内存，这种错误可能有很多情况，看下面的例子。

```
#define array_size 10
int *a = new int[array_size];
... ...
int *b = new int[array_size];
... ...
memcpy(b, a, array_size);
```

这个错误可能较数组越界访问更为隐蔽。程序的原意是对两个整型数组进行复制，但上面的代码仅仅复制了其中array_size个字节，而事实上程序要完成既定任务，用于指示内存空间大小的参数应该为array_size * sizeof(int)个字节。

再举一个分配空间不足的例子。有时程序员会因为粗心而忘记字符串最后需要一个结束符“ \0 ”，如果忘记了这一点就有可能引起错误，考虑下面一段代码。

```
char *new_string(char *s)
{
    int len = strlen(s);

    char *new_s = new char[len];
    strcpy(new_s, s);
    return new_s;
}
```

上述例子中的错误在于程序没有为 string 分配足够的空间，用于指示新字符串长度的参数应该是 len+1，这样才给结束符留了一个字节。

此外，很多字符串操作并不做检查以确保字符串能够被适当地写进被分配的内存中，这又给超量写内存错误的发生提供了可能。例如，当两个字符串被连接时，结果字符串可能占用更多的空间。

```
char s1[] = "Hello";
char s2[] = " world";
char s[5];
strcpy(s, s1);
```

```
strcat(s, s2);
```

连接字符串的长度是 11 个字符，它占据了 12 字节（包括最后的结束符），但只有 5 字节的空间被分配。连接操作将在字符串 s 结束后继续进行写操作。这显然修改了其他数据。

操作符的优先权也是迷惑程序员并诱使他们犯错的一个常见因素，在本章前面关于指针运算的介绍中强调过这一点，不慎地使用*和++（或– –），都有可能导致超量写内存。

另一个可能的错误是构造一个指向运行时栈中某值的指针，并将这个指针返回给函数调用者，例如：

```
int *p ()
{
   int i = 0;
   return &i;
}
```

尽管函数返回了一个指向零值整型的指针，但由于该整型变量位于一个活动记录中，而这个记录在函数返回时就会被删除。因此这个指针指向的内容随时会变成任何一个值，这完全取决于下一个使用它的函数如何操作它。

总之，构造错误指针的可能有很多种，它们都会在不应当写数据的地方进行写操作。在 C++程序中，这些错误都非常常见，而且它们十分隐蔽，很难被发现。即使它们被发现，或许已经太迟了，最好的途径应当是养成良好的编程习惯以尽量避免它们的发生。

本章参考文献

[1]　左飞. 代码揭秘 :从 C/C++的角度探秘计算机系统. 北京, 电子工业出版社, 2009.

[2]　Bruce Eckel. 刘宗田等译. C++编程思想（第 2 版）. 北京, 机械工业出版社, 2002.

第 3 章

字符串与模式匹配——梦里寻她千百度

字符串类型可以看作是特殊的数组——是一种由字符型变量组成的数组。字符串处理在现实应用中大量存在。比如用户的姓名、家庭住址、文章的标题等都以字符串形式存在。另一方面，在数据结构与算法这个框架下，关于字符串的匹配操作也有很多话题值得讨论。这部分内容尽管显得有些晦涩深奥，但也因闪烁着智慧的光芒而富于妙趣。

3.1 基本概念与定义

文本内容的输入和输出是计算机最常进行的操作之一。字符串是一种非常方便且常用的数据类型，它提供了一种进行文本操作的途径。本节介绍有关字符串的相关内容。

3.1.1 C++中的字符串

字符串是n个字符的有限序列，其中 $n \geqslant 0$，例如“ Hello World! ”。这些字符在内存中按顺序逐个存放，并以结束符“ \0 ”结尾，如图 3-1 所示。如果读者通过前面的学习对于数组已经有了一个清晰的认识，那么完全可以把字符串看作是一种特殊的数组，其中每个元素都是一个字符型变量。由此出发，字符串就已经非常容易理解和掌握了。

求职面经点拨

请试回答附录题库中的第 17 题。

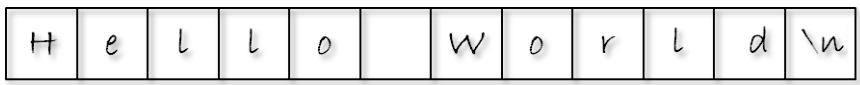


图 3-1 字符串

标准 C++库中预定义了 string 类，并为这种数据类型提供了必要的接口函数来支持字符串的一些基本操作。字符串也是一种典型的线性数据结构。但是 C++的 string 类型并不是对于所有程序都默认可用的。如果一个 C++程序需要 string 类型，程序员必须引用定义这种类型的库 string。类 string 封装了字符串的属性并提供了一系列允许访问这些属性的服务。下面对该类中描述的有关字符串的操作进行简要介绍，首先 string 类的操作符定义如表 3-1 所示。

表 3-1　string 类的操作符

操 作 符	示　　例	说　　明
+	s+t	将字符串 s 和字符串 t 连接成一个新字符串
=	s=t	用字符串 t 更新字符串 s
+=	s+=t	作用同 s = s+t
==	s==t	判断字符串 s 和字符串 t 是否相等
!=	s!=t	判断字符串 s 和字符串 t 是否不相等
<	s<t	判断字符串 s 是否小于字符串 t
<=	s<=t	判断字符串 s 是否不大于字符串 t
>	s>t	判断字符串 s 是否大于字符串 t
>=	s>=t	判断字符串 s 是否不小于字符串 t
[]	s[i]	访问字符串中下标为 i 的字符

由于 string 类中的成员函数非常多，且大都具有多种重载形式，因此不能一一尽述，这里只选择其中较常用的几个进行介绍。

- string append(const char * s);

求职面经点拨

请试回答附录题库中的第 18 题。

该函数的作用是将字符串 s 追加到本字符串的末尾。

- string assign(const char * s);

该函数的作用是将字符串 s 赋给本字符串。

- int compare(const string& str)const;

该函数的作用是比较两个字符串是否相同。

- string& insert(unsigned int p0, const char * s);

该函数的作用是将字符串 s 插入到本字符串 p0 位置之前。

- string substr(unsigned int pos, unsigned int n)const;

该函数的作用是取出本字符串中位置 pos 开始的 n 个字符，并将该新字符串返回。

- unsigned int find(const basic_string& str)const;

该函数的作用是查找子字符串 str 在本字符串中第一次出现的位置。

- unsigned int length()const;

该函数的作用是获得本字符串的长度。

- void swap(string & str);

该函数的作用是将本字符串与字符串 str 进行交换。

为了方便读者熟悉和掌握以上函数，下面给出了一个示例程序。这个示例程序首先接收一个含有 3 个空格的字符串作为输入。然后将这个字符串按空格进行拆分，最终获

得4个独立的新字符串再逐个输出。然后将得到的4个字符串进行重组，再与原字符串进行比较并判断是否相同。

```
#include <iostream>
#include <iomanip>
#include <string>

using namespace std;

int main(int argc, char** argv) {

    string  str = "高德纳 美国 计算机科学家 计算机程序设计艺术";
    string  str_temp = "";
    str_temp.assign(str);

    string result[4]={"","","",""};

    int position = 0;

    for(int i = 0; i < 3; i++)
    {
        position = str_temp.find(" ");
        result[i] = str_temp.substr(0,   position);
        str_temp = str_temp.substr(position + 1, str_temp.length() - position);
    }

    result[3] = str_temp;

    cout<<"姓名："<<setw(8)    <<result[0]<<endl;
    cout<<"国籍："<<setw(6)    <<result[1]<<endl;
    cout<<"职业："<<setw(14)   <<result[2]<<endl;
    cout<<"代表作："   <<setw(18)   <<result[3]<<endl;

    str_temp.swap(result[0]);

    for(int j = 1; j < 4; j++)
    {
        str_temp+=" ";
        str_temp.append(result[j]);
    }

    int equal = str.compare(str_temp);

    if(equal==0)
    {
        cout<<"Successful Matching！"<<endl;
    }
    else
        cout<<"Unsuccessful Matching！"<<endl;

    return 0;
}
```

完成编码后，编译并运行程序即可。

3.1.2　字符串抽象数据类型

求职面经点拨

请试回答附录题库中的第 19 题。

字符串是简单的线性数据结构，通过前面的介绍，可以知道它是由字符元素顺序排列而成的线性表。它的结构相对简单，且具有与其他线性结构相似的操作函数，例如字符串的初始化和元素的查找等。但由于其操作对象比较特殊且相对固定，因而字符串结构也有其特有的一些函数。首先给出字符串抽象数据类型的类定义如下。

```
const int maxLen = 128;
class String {
    int curLen;          //字符串的长度
    char * c;            //字符串的存储数组
public:
    //构造函数
    String ( const String& ob );
    String ( const char * init );
    String ( );
    //析构函数
    String ( ) {delete [] c;}

    int Length ( ) const {return curLen;}   //求当前字符串的实际长度
    int Find ( String& pat ) const;         //在当前字符串中查找目标字符串

    //取当前字符串中从 pos 开始的 len 个字符组成的子字符串
    String &operator ( ) ( int pos, int len );
    //判断当前字符串与对象串 ob 是否相等
    int operator == ( const String &ob ) const {
                     return strcmp (ch, ob.ch) == 0; }
    //判断当前字符串与对象串 ob 是否不等
    int operator != ( const String &ob )const {
                     return strcmp (ch, ob.ch) != 0; }

    int operator ! ( )const { return curLen == 0; } //判断当前字符串是否为空
                                                    //字符串
    String &operator = (String &ob);   //将字符串 ob 赋给当前字符串
    String &operator += (String &ob);  //将字符串 ob 连接到当前字符串之后
    char &operator [ ] ( int i );      //取当前字符串的第 i 个字符
};
```

下面给出字符串类中 3 个构造函数的实现代码。首先是构造函数 1，该函数从已有字符串 ob 中复制得到新字符串。

```
String :: String ( const String& ob )
{
    c = new char[maxLen+1];        //创建字符串数组
    if ( c == NULL ) {
        cerr << “存储空间分配错误! \n”;
        exit(1);
    }
    curLen = ob.curLen;            //复制字符串长度
    strcpy ( c, ob.c );            //复制字符串值
}
```

构造函数 2，该函数从已有字符数组*init 复制得到新字符串。

```
String :: String ( const char *init )
{
    c = new char[maxLen+1];    //创建字符串数组
    if ( c == NULL ){
        cerr << “存储空间分配错误! \n”;
        exit(1);
    }
    curLen = strlen ( init );  //复制字符串长度
    strcpy ( c, init );        //复制字符串值
}
```

构造函数 3，该函数创建一个空字符串。

```
String :: String ()
{
    c = new char[maxLen+1];   //创建字符串数组
    if ( ch == NULL ) {
        cerr << “存储空间分配错误!\n”;
        exit(1);
    }
    curLen = 0;
    c[0] = ‘\0’;
}
```

类中的其他函数此处不再详细列出，有兴趣的读者可以尝试自行实现。特别是其中关于字符串搜索的函数，读者完全可以在本章后续内容的基础上，尝试完成数个不同的版本。

3.2 文本的精确匹配

字符串是一种线性表，它在计算机应用系统（如文本编辑、情报检索、拼写检查、互联网搜索引擎和自然语言翻译等）中有着广泛的应用。而在计算机科学领域，模式匹配问题一直都是学者们研究和关心的热点之一。模式匹配是指在目标文本串$T(t_0,t_1,t_2,t_3,\cdots,t_{n-1})$中检索子串$P(p_0,p_1,p_2,p_3,\cdots,p_{m-1})$的过程，其中$P$称为模式（Pattern）。若能够在$T$中找到子串$P$，则称匹配成功，否则匹配失败。模式匹配技术广泛应用于入侵检测、全文检索、计算机病毒特征码匹配、数据压缩以及 DNA 序列匹配等领域中，匹配算法的好坏是直接影响该系统效率的关键。

3.2.1 BF 算法

算法

BF 算法,很简单、很暴力!

BF（Brute Force）算法是模式匹配中最简单、最直观的算法。该算法的基本思想是从主串$T(t_0,t_1,t_2,t_3,\cdots,t_{n-1})$中的第start个字符起和模式的第 1 个字符比较，如果相等，则继续逐个比较后续字符；比较过程中一旦发现不相等的情况，则回溯至主串中的第start + 1个字符位置处，重新和模式P的字符进行比较。依此类推，直到模式P中的每个字符依次和主串T中的一个连续的字符序列完全相等，则匹配成功，否则匹配失败。通过分析易知该算法的复杂度为$O[(n-m)m]$，其特点是直观、简单，但由于涉及多次回溯，算法效率很低。对于信息量较大情况下的模式匹配不适用。

举一个简单的例子来说明 BF 算法到底是如何工作的。如图 3-2 所示，这里的匹配任务是在字符串$T(CAATCAT)$中搜索是否存在模式$P(CAT)$。算法首先将t_0与p_0进行比较，发现二者相同；于是继续比较t_1和p_1，结果表明二者仍相同；于是继续比较t_2和p_2，结果发现二者不同。这时算法将模式串P向右滑动一个位置，并比较p_0与t_1，结果表明二者不同；于是算法继续将模式串P向右滑动一个位置，并比较p_0与t_2，结果表明二者仍然不同；同理算法继续将模式串P向右滑动并进行比较，直到发现p_0与t_4、p_1与t_5、p_2与t_6均能一一对应，完全相同时，则匹配成功。

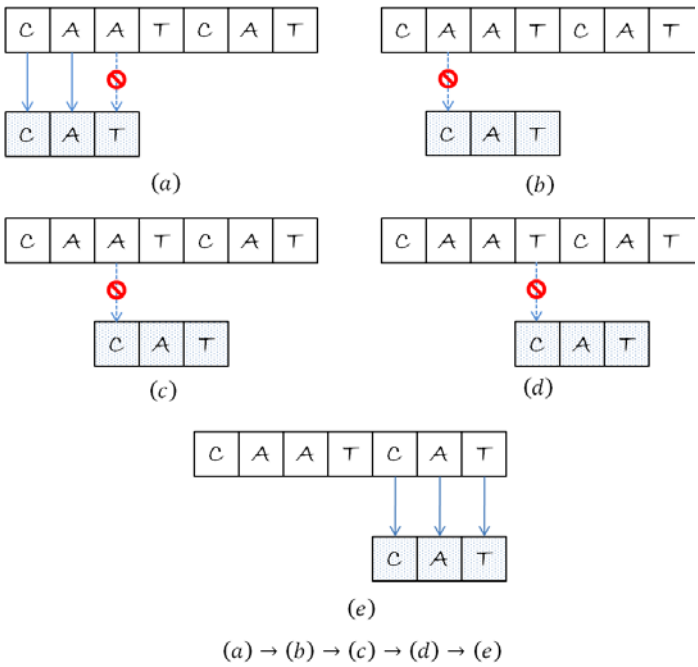


图 3-2　BF 算法样例

3.2.2　MP 算法

算法

MP 算法，它与 KMP 算法很像，但被提出时间要更早些。

詹姆斯 · 莫里斯（James Morris）和沃恩 · 普莱特（Vaughan Pratt）在 1970 年提出了一种快速串匹配算法——MP 算法。它是对 BF 算法的很大改进。这主要体现在每当某趟匹配失败时，指针不必回溯，而是利用已经得到的“部分匹配”结果，将模式向右“滑动”若干位置后继续比较。由于 MP 算法避免了 BF 算法中频繁的回溯，普遍提高了模式匹配的工作效率，因此它又被称为不回溯的字符串搜索算法。

不能有效地利用已经得到的比较结果，使过去的工作没有最大程度地发挥效用，这是造成 BF 算法频繁回溯、多次比较的主要原因。如果能把已经得到的比较结果“储存”起来，并在后续比较过程中直接调用这些既得结果，必然会大大提高效率，这就是 MP 算法的基本思想。为了更加清楚明了，下面结合实例介绍 MP 算法的工作方式。

假设有目标串$T(t_0,t_1,t_2,t_3,\cdots,t_{n-1})$和模式串$P(p_0,p_1,p_2,p_3,\cdots,p_{m-1})$，若使用 BF 算法进行模式匹配，当进行第 1 轮比较时，若$t_k \neq p_k$，则算法结束本轮比较，如图 3-3 所示。

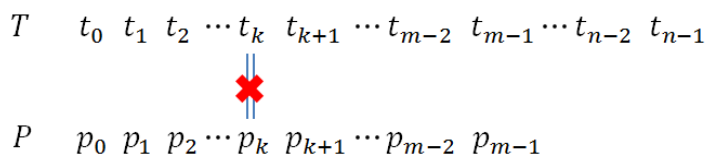


图 3-3　第 1 轮比较结束

由于字符串T与P中第 1 个不相等的字符出现在位置k处，所以两字符串前k个字符是相等的。此时，可用字符串$P(p_0,p_1,p_2,p_3,\cdots,p_{k-1})$代替字符串$T'(t_0,t_1,t_2,t_3,\cdots,t_{k-1})$，于是原目标串可转化为$T(p_0,p_1,p_2,p_3,\cdots,p_{k-1},t_k,\cdots,t_{n-1})$。在进行第 2 轮比较之前，算法同样把字符串$P$整体向后移动一个字符，此时字符串$T$与$P$之间的关系如图 3-4 所示。

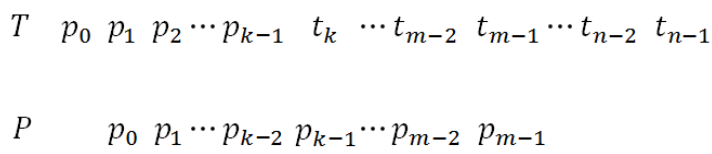


图 3-4　第 2 轮比较

在第 2 轮比较中，算法先要比较的字符是P中首字符p_0与T中第 2 个字符p_1，若p_0与p_1相等，则算法顺序比较P中第 2 个字符p_1与T中第 3 个字符p_2，若不等，则算法仍然把模式串P整体向后移动一个字符，此时字符串T与字符串P之间的关系如图 3-5 所示。

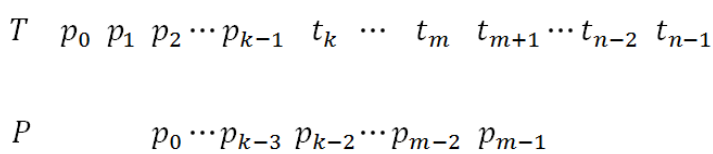


图 3-5　第 3 轮比较

算法依照同样的次序，首先对P中字符p_0与T中字符p_2进行比较，若相等则顺序比较后续的字母，若不等则再把字符串P整体向后移动一个字符。

细心的读者可能会发现，在第 2 轮比较开始时，首先进行比较的字符是p_0与p_1，其次进行比较的字符是p_1与p_2；在第 3 轮比较开始时，首先进行比较的字符是p_0与p_2，其次进行比较的字符是p_1与p_3。而p_0、p_1、p_2、p_3全部是字符串P中的字符，它们之间的关系完全可以在调用字符串匹配算法之前就确定下来。

MP 算法正是利用这种思想，在对字符串进行匹配之前，先计算出模式字符串（字符串P）中各个字符间的关系，然后再依据此关系对模式字符串与目标字符串（字符串T）进行匹配。在上述过程中，记录字符串P中各个字符之间关系的函数也被称为字符串P

的失效函数。下面通过一个具体的例子来介绍获取失效函数的方法。

假设模式字符串P为$caatcat$，该如何计算它的失效函数呢？换句话说，字符串P众多字符之间的关系中，哪些关系是 MP 算法所需要的？针对以上问题，下面将分步来讲述失效函数的获取过程。

众所周知，函数由定义域、值域与映射组成。那么什么是失效函数的定义域呢？失效函数自变量j的取值范围是模式字符串在“ 失配 ”前匹配的字符个数。例如，模式字符串P，由于不知道字符串T的具体情况如何，所以P可能在任何位置上“ 失配 ”。例如，P在第 1 个字母c处就发生失配，则j的取值为0，因为在“ 失配 ”前没有匹配字符；若P在第 6 个字符a处“ 失配 ”，由于a前的 5 个字母一定匹配成功，所以j的取值可以是5。综上所述，在模式$P = caatcat$中，失效函数的定义域为$j \in \{0,1,2,3,4,5,6\}$。可见，失效函数的定义域为$0 \sim Len(P) - 1$，其中$Len(P)$为模式字符串P的长度。

在确定了失效函数的定义域后，下面考察失效函数值域的计算方法。失效函数的取值k定义为$k \in \{x|0 \leqslant x < j\}$，且$k$是满足条件$p_0 p_1 \cdots p_k = p_{j-k} p_{j-k+1} \cdots p_j$的最大正整数。当然这样的$k$有可能并不存在，此时规定失效函数的取值为$-1$。下面就利用以上规则求解字符串$P$的失效函数值域。

当$j = 0$时，由于$0 \leqslant k < 0$，所以满足条件的k并不存在，失效函数的取值为-1，则有$f(0) = -1$；当$j = 1$时，k可能的取值为0。由于$p_0 \neq p_1$，所以k不能取0，满足条件的k不存在，失效函数的值仍为-1，即$f(1) = -1$；当$j = 2$时，k可能的取值为0或1。由于$p_0 \neq p_2$且$p_0 p_1 \neq p_1 p_2$，所以满足条件的k不存在，$f(2) = -1$；当$j = 3$时，k可能的取值为0、1或者2。由于$p_0 \neq p_3$、$p_0 p_1 \neq p_2 p_3$及$p_0 p_1 p_2 \neq p_1 p_2 p_3$，所以满足条件的$k$不存在，即$f(3) = -1$。同理当$j = 4$时，$k$可能的取值为0、 1、2、3。由于$p_0 = p_4$、$p_0 p_1 \neq p_3 p_4$、$p_0 p_1 p_2 \neq p_2 p_3 p_4$以及$p_0 p_1 p_2 p_3 \neq p_1 p_2 p_3 p_4$，所以满足条件的$k$为0，此时$f(4) = 0$。当$j = 5$时，$k$可能的取值为0、1、 2、3、 4。由于$p_0 \neq p_5$、$p_0 p_1 = p_4 p_5$、$p_0 p_1 p_2 \neq p_3 p_4 p_5$、$p_0 p_1 p_2 p_3 \neq p_1 p_2 p_3 p_4$以及$p_0 p_1 p_2 p_3 p_4 \neq p_1 p_2 p_3 p_4 p_5$，所以$f(5) = 1$。同理可以求得，当$j = 6$时，$f(6) = -1$。最终模式字符串$P = caatcat$的失效函数如表 3-2 所示。

表 3-2　失效函数示例

j	0	1	2	3	4	5	6
$p(j)$	c	a	a	t	c	a	t
$f(j)$	–1	–1	–1	–1	0	1	–1

得到模式字符串P的失效函数后，就可以应用 MP 算法对它进行匹配。具体的匹配过程分两种情况。假设在进行某一轮比较时，失配的情况发生在模式P的第j位，如果$j = 0$，则让目标的指针前进一位，模式的起始比较地址回到p_0。否则，在进行下一轮比较时，目标指针不发生回溯，仍指向失配的位置，而模式P的起始比较地址为$p_{f(j-1)+1}$。由失效函数的计算过程可见，函数$f(j)$仅与字符串P有关，而与目标字符串（字符串T）无关。所以，只要给定模式字符串P，无论目标字符串T的取值是什么，均可应用同一个

失效函数对它们进行匹配。

现举例说明 MP 算法是如何利用失效函数来对字符串进行匹配的。这里假设模式字符串$P = caatcat$，目标字符串$T = ctcaatcacaatcat$。在第 1 轮匹配前，首先把模式字符串P与目标字符串T从各自第 1 个字符起对齐，以便进行比较，如图 3-6 所示。

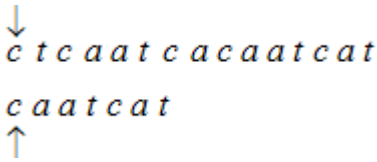


图 3-6　第 1 轮匹配

由第 1 轮比较的结果可见，模式字符串与目标字符串在第 2 个字符处发生失配。算法检测到失配后结束本轮比较，并且目标指针不发生回溯，仍指向失配的位置。由于失配发生在第 2 个字符处，此时$j = 1$。所以模式P在下一轮匹配时的起始地址为$p_{f(1-1)+1}$，即p_0。两字符串之间的关系如图 3-7 所示，此时可进行第 2 轮比较。

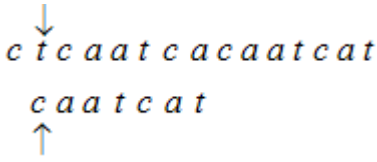


图 3-7　第 2 轮匹配

第 2 轮比较中，由于模式字符串P在的第 1 个字符处发生失配，此时$j = 0$，所以让目标的指针前进一位，模式的起始比较地址回到p_0。两字符串之间的关系如图 3-8 所示，此时可进行第 3 轮比较。

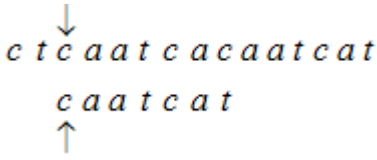


图 3-8　第 3 轮匹配

发现在模式字符串P中的第 7 个字符处发生失配，此时$j = 6$。可知模式字符串P在下一轮匹配时的起始比较地址为$p_{f(6-1)+1}$，即p_2。目标指针同样不发生回溯，仍指向失配的位置，如图 3-9 所示，此时可进行第 4 轮匹配，经过第 4 轮比较，匹配成功。

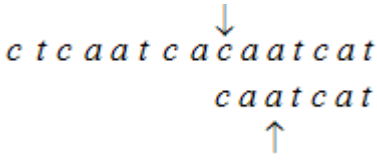


图 3-9　第 4 轮匹配

通过简单的分析易知，假设模式字符串P的长度为m，目标字符串T的长度为n，那么 MP 算法的时间复杂度为$O(m+n)$。

下面给出 MP 算法的实现代码。读者也可从本书的在线支持网站中下载得到用于验证下列函数使用方法的测试程序。

```
void preMp(const char *x, int m, int mpNext[]) {
    int i, j;
    i = 0;
    j = mpNext[0] = -1;
    while (i < m) {
        while (j > -1 && x[i] != x[j])
            j = mpNext[j];
        mpNext[++i] = ++j;
    }
}

void MP(string p, string t) {

    int m = p.length();
    int n = t.length();
    if (m > n) {
        cerr<<"Unsuccessful match!"<<endl;
        return;
    }
    const char * x = p.c_str();
    const char * y = t.c_str();

    int i = 0, j = 0, mpNext[m+1];
    preMp(x, m, mpNext);

    while (j < n) {
        while (i > -1 && x[i] != y[j])
            i = mpNext[i];
        i++;
        j++;
        if (i >= m) {
            cout<<"Matching index found at: "<<j - i<<endl;
            i = mpNext[i];
        }
    }
}
```

最后我们对以上代码稍微做一些解释。对于给定的模式字符串$P = caatcat$，可以执行一下函数 preMp()，执行的结果会为数组 mpNext[]赋值，其取值结果如表 3-3 所示。

表 3-3　mpNext 表

j	0	1	2	3	4	5	6	7
$p(j)$	c	a	a	t	c	a	t	
mpNext[j]	-1	0	0	0	0	1	2	0

前面已经讲过，当失配发生在模式字符串P的第j位且$j \neq 0$时，则模式字符串P的起始比较地址为$p_{f(j-1)+1}$。所以很容易算得，当$j = 6$时，$p_{f(6-1)+1} = p_2$。当$j = 5$时，$p_{f(5-1)+1} = p_1$。据此可以一直推算到当$j = 1$时，$p_{f(1-1)+1} = p_0$。可见，数组 mpNext[]存储的是失配发生时，进行下一轮比较过程中模式P的起始比较地址。

但还有两个比较特殊的值需要引起注意。其一是$j=0$的情况，正如前面所讲过的，此时需令目标字符串的指针前进一位，模式的起始比较地址回调到p_0。由于目标字符串的指针前进了一位，结合后续 MP 函数中的代码实现，所以 mpNext[0]的值是-1。另外还应该注意到$j=7$的情况，此时可以算出$p_{f(7-1)+1}=p_0$，现在的问题是尽管这个值可以被计算出来，但它是否有意义呢？显然，在上例中当$j=7$时其实表明已经找到了一个匹配。事实上，如果目标字符串仍然有后续部分，这个匹配工作仍然可以继续进行。也就是直至找出目标文本中所有匹配的字符串为止。因此，如果不是要仅找出模式字符串首次出现的位置，那么 mpNext[7]这个值就是有意义的。上面给出的函数执行的结果就是输出目标文本中所有匹配成功的子串的位置。

算法

KMP 算法，在某种程度上，克努特的名气也间接地扩大了这个算法的影响力。此外，IT 公司笔试中也常会考到这个算法。

克努特是算法和程序设计技术的先驱，他还发明了计算机排版系统 TeX 和字体设计系统 Metafont。他的巨著《计算机程序设计的艺术》誉满全球，被认为是 20 世纪最重要的 12 本科学专著之一，与爱因斯坦的《相对论》比肩而立。1974 年，克努特荣获计算机领域的最高奖——图灵奖，彼时他只有 36 岁。目前他仍然是最年轻的图灵奖获得者这一纪录的保持者。

3.2.3 KMP 算法

KMP 算法是由唐纳德·克努特（Donald Knuth）、莫里斯和普莱特于 1977 年共同提出的快速串匹配算法。可以说它与前面介绍的 MP 算法一脉相承，二者的设计初衷都是试图充分利用先前匹配过程中已经得到的结果来避免频繁回溯。由于这两种算法非常相似，所以也导致很多人常常将二者混为一谈。

先来回顾一下 MP 算法的设计思路。如图 3-10 所示，当前模式字符串P的左端的p_0与目标字符串中的T_j位置对齐。从左向右逐个进行比较，发现p_i处的字符a与T_{j+i}处的字符b发生失配。同时也表明，$P(p_0,p_1,\cdots,p_{i-1})$与$T'(t_j,t_{j+1},\cdots,t_{j+i-1})$是完全匹配的，这一部分子串在图中用字母$u$标示出。由于发生失配，随即移动模式字符串并进行下一轮的比较。此时，很自然地希望移动之后的结果可以使得模式字符串P中的一个前缀v，可以匹配到子串u的某一部分后缀。MP 算法的初衷就是找到像这样的一个能够匹配u中某一后缀的最长的前缀v。正如 3.2.2 节所做的那样，我们引入了一个标记mpNext[i]，并用它来对P中符合要求的最长前缀进行标记。于是在模式字符串的一次移动后，新一轮的比较即可在字符$P_{\text{mpNext}[i]}=c$和$T_{i+j}=b$之间展开。特别地，我们将其中 mpNext[0]的值置为-1。利用这种思想，MP 算法跳脱了 BF 算法中每次仅能将模式串向右移动一个字符位置的窘境。

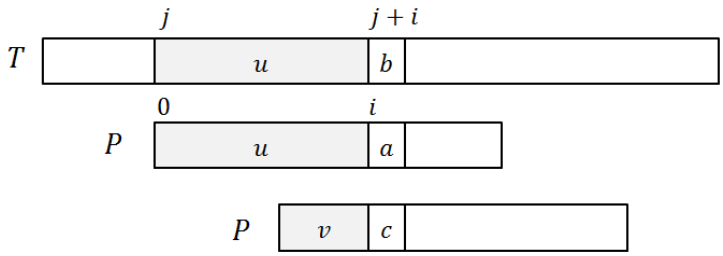


图 3-10　模式串的偏移

现在我们希望在 MP 算法的基础上再推进一步。继续前面的过程，当模式字符串P完成一次移动之后，接下来马上要进行的工作就是比较字符b和c。要是可以有效避免b和c之间能够预见的一次失配，显然就将原来的算法又向前推进了一步。当然，在比较之前我们并不能想当然地预见到字符b，因为我们手头仅用的材料是模式字符串P。但要

想尽量避免随之而来的一次失配，在仅仅知道模式字符串P的情况下，我们至少可以保证完成一次移动之后，紧随前缀字符串v之后的那个字符c不等于原来失配的字符a。这显然会提升匹配的成功率。有时也称满足上述条件的最长的前缀v是字符串u的加标边际（Tagged Border）。

KMP 算法使用kmpNext表来对P中符合要求的加标边际进行标识。注意这里所说的“符合要求”，是指同时符合两个要求。首先，v是可以匹配到u中某个后缀的一个最长前缀。其次，仅跟在v后面的字符c不同于紧跟在u后面的字符a。执行 MP 算法所得到的mpNext表已经实现了求最大公共子串这一任务。于是不妨设想在此基础之上，继而将“紧跟在v后面的字符不同于紧跟在u后面的字符”这一条件也引入。生成mpNext表的方法 3.2.2 节中已经给出了详细的介绍，这里不再赘述。下面就在mpNext表已知的情况下，给出建立kmpNext表的规则。这个规则被分为 4 种情况，其中$1 \leqslant j \leqslant m-1$：

1）如果mpNext[j] $=0$且$p_j = p_0$，则令kmpNext[j] $=-1$。

2）如果mpNext[j] $=0$且$p_j \neq p_0$，则令kmpNext[j] $=0$。

3）如果mpNext[j] $\neq 0$且$p_j \neq p_{\text{mpNext}[j]}$，则令kmpNext[$j$] $=$ mpNext[j]。

4）如果mpNext[j] $\neq 0$且$p_j = p_{\text{mpNext}[j]}$，则用mpNext[$j$]的值替换原来mpNext[$j$]中的$j$值，直到情况转换为前 3 种情况中的一种，进而递归地求解kmpNext[j]。

此外，同样将kmpNext[0]的值置为-1，并令kmpNext[m] $=$ mpNext[m]，其中m是模式串P的长度。这一举措的目的与 MP 算法相同，即当算法找到一个成功的匹配之后，后续目标字符串并未完结，kmpNext[m]的值就指示了为继续进行后续匹配而需将模式字符串移动的位数。

仍然以模式字符串$P = caatcat$为例并按照3.2.2节中介绍的方法建立相应的mpNext表，如表3-4所示。不难发现，$j=4$符合第1种情况。因为mpNext[4] $=0$且$p_4 = c = p_0 = c$，所以mpNext[4] $=-1$。而$j=1,2,3$则符合第 2 种情况。比如，当$j=1$时，mpNext[1] $=0$且$p_1 = a \neq p_0 = c$。另外，$j=6$属于第 3 种情况。此时mpNext[6] $=2 \neq 0$，$p_6 = t \neq p_2 = a$，所以kmpNext[6] $=$ mpNext[6] $=2$。最后，$j=5$属于第 4 种情况。此时mpNext[5] $=1 \neq 0$且$p_5 = a = p_1 = a$。为此，用mpNext[5]的值替换原来mpNext[j]中的j值，即令$j=1$，于是得到mpNext[j] $=0$且$p_{j=1} \neq p_0$。此时情况转化为第 2 种类型，所以kmpNext[5] $=0$。最终所得的结果如表 3-4 所示。

求职面经点拨

请试回答附录题库中的第 20 题。

表 3-4　kmpNext表

j	0	1	2	3	4	5	6	7
$p(j)$	c	a	a	t	c	a	t	
mpNext[j]	–1	0	0	0	0	1	2	0
kmpNext[j]	–1	0	0	0	–1	0	2	0

有必要来总结一下kmpNext[j]中各值的意义。

（1）对于kmpNext[0] $=-1$，这是因为前面已经规定任何模式字符串的首字符对应的

kmpNext[j]值都为-1。

（2）对于kmpNext[j] $= -1$，表示模式字符串P中，字符p_j与首字符相同，且p_j前面k个字符与模式字符串开头的k个字符不相等，或者相等但$p_j \neq p_k$，其中$j \neq 0$，$1 \leqslant k < j$。

（3）对于kmpNext[j] $= k$，表示模式字符串P中，字符p_j前面k个字符与模式字符串开头的k个字符相等，即$p_0 p_1 \cdots p_{k-1} = p_{j-k} p_{j-k+1} \cdots p_{j-1}$，且$p_j \neq p_k$，其中$1 \leqslant k < j$。

（4）除上述 3 种情况以外，kmpNext[j] $= 0$。

表 3-5 和表 3-6 所给出的两个例子留给读者用作练习，此处不再做过多的解释。

表 3-5　kmpNext表（思考练习 1）

j	0	1	2	3	4	5	6	7	8
$p(j)$	a	b	c	a	b	c	a	d	
mpNext[j]	–1	0	0	0	1	2	3	4	0
kmpNext[j]	–1	0	0	–1	0	0	–1	4	0

表 3-6　kmpNext表（思考练习 2）

j	0	1	2	3	4	5	6	7	8
$p(j)$	a	d	c	a	d	c	a	d	
mpNext[j]	–1	0	0	0	1	2	3	4	5
kmpNext[j]	–1	0	0	–1	0	0	–1	0	5

得到kmpNext表之后，即可以此为基础对字符串进行模式匹配。具体而言，每次发生失配后，模式字符串向右移动的位数由$j -$ kmpNext[j]给出。比较过程与 MP 算法非常类似，这里不再赘述。

最后给出 KMP 算法的实现代码。读者同样可以从本书的在线支持网站中下载得到用于验证下列函数使用方法的测试程序。

```
void preKmp(const char *x, int m, int kmpNext[]) {
    int i, j;

    i = 0;
    j = kmpNext[0] = -1;
    while (i < m) {
        while (j > -1 && x[i] != x[j])
            j = kmpNext[j];
        i++;
        j++;
        if (x[i] == x[j])
            kmpNext[i] = kmpNext[j];
        else
            kmpNext[i] = j;
    }
}

void KMP(string p, string t) {

    int m = p.length();
    int n = t.length();
```

```
    if (m > n){
        cerr<<"Unsuccessful match!"<<endl;
        return ;
    }

    const char * x = p.c_str();
    const char * y = t.c_str();

    int i = 0, j = 0, kmpNext[m+1];
    preKmp(x, m, kmpNext);

    i = j = 0;
    while (j < n) {
        while (i > -1 && x[i] != y[j])
            i = kmpNext[i];
        i++;
        j++;
        if (i >= m) {
            cout<<"Matching index found at: "<<j - i<<endl;
            i = kmpNext[i];
        }
    }
}
```

3.2.4　BM 算法

算法

BM 算法，运用逆向思维突破传统束缚的经典范例。

据说在发明吸尘器之前，人类已经发明了吹尘器。从吹到吸，无疑是逆向思维成功运用的典范。逆向思维往往能够让人从原有的思维定式中跳脱出来，打破既有沉珂的藩篱，因此成为驱动创新的一种神兵利器。KMP 算法可以说在原有范式的框架下已经将模式匹配算法的效率发挥到了极致。若要将算法的时间效率再做进一步提升，就必须要打破原有的思维定式，找出一条非比寻常的道路。不妨运用逆向思维来重新考虑问题。在过往算法中，模式字符串与目标字符串的比较都是从左向右进行的。如果将比较过程从右向左进行能否开创一个崭新的局面呢？

1977 年，美国计算机科学家罗伯特 · 波义耳（Robert Boyer）和杰 · 摩尔（J · Moore）共同提出了著名的 BM 算法。使用 BM 算法进行模式匹配时，模式字符串$P(p_0,p_1,p_2,p_3,\cdots,p_{m-1})$由左向右移动，而字符的比较却自右向左进行，即按$P(p_{m-1},p_{m-2},p_{m-3},\cdots,p_0)$的次序进行比较。当文中字符与模式不匹配时，则根据预先定义好的“坏字符偏移函数”和“好后缀偏移函数”计算出偏移量。

坏字符偏移和好后缀偏移这两个函数的原理可以简单描述如下：当比较工作进行到文中的j处时，新一轮的比较从右向左进行。不相匹配的情况出现在模式字符串的$P[i]=a$以及目标字符串的$T[i+j]=b$处，那么这时有$T[i+j+1,\cdots,j+m-1]=P[i+1,\cdots,m-1]$且$P[i]\neq T[i+j]$。

由好后缀偏移函数确定的偏移量分两种情况。首先，如图 3-11 所示，将模式字符串与目标字符串进行从右向左比较，当比较到字符$P[i]$时发现不同，则把$T[i+j+1,\cdots,j+m-1]=P[i+1,\cdots,m-1]$这一段看作是$u$。如果在模式字符串里面$u$再次出现，且$u$的左侧是一个不同于$a$的前缀$c$时，则将模式字符串最右端再次出现的且前缀不是$a$的片

段与文本串$T[i+j+1,\cdots,j+m-1]$对齐，并将这次移动的偏移量存储在bmGs[i]中。

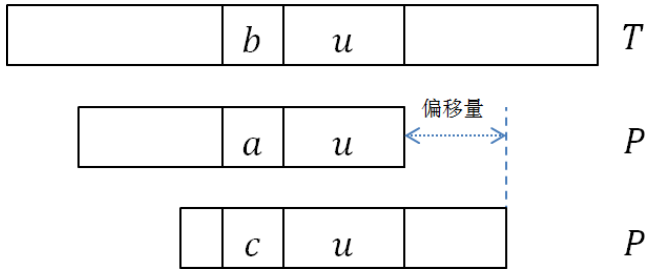


图 3-11　好后缀偏移函数情况 1

其次，如果u在模式中没有再次出现，或再次出现但u前面字符是a，那么就将片段$T[i+j+1,\cdots,j+m-1]$中最长的后缀v与模式字符串前缀v对齐，如图 3-12 所示。若不存在这样的v，则模式字符串跳过u，并将这次移动的偏移量存储在bmGs[i]中。

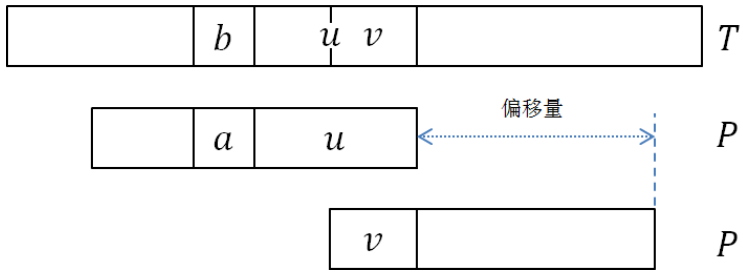


图 3-12　好后缀偏移函数情况 2

坏字符偏移函数的移动过程也分两种情况。首先，如图 3-13 所示，将目标字符串中的字符b同它在P $[0,\cdots,m-2]$中从右向左首次出现的位置对齐，并将这次移动的偏移量存储在bmBc[b]中。

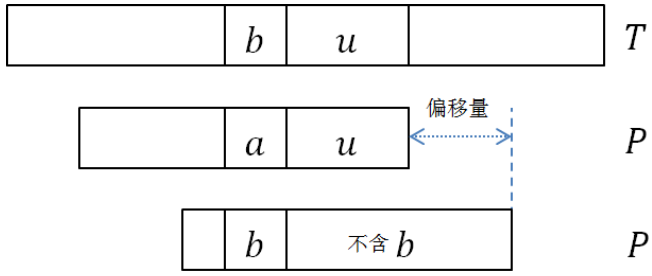


图 3-13　坏字符偏移函数情况 1

其次，如图 3-14 所示。如果在模式字符串中没有b，则将窗口的左端点与$T[i+j+1]$对齐，同理得到偏移量。

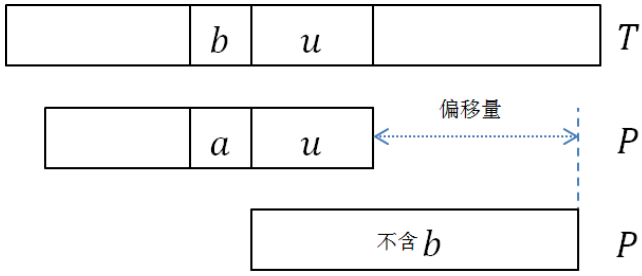


图 3-14　坏字符偏移函数情况 2

无论是坏字符偏移函数还是好后缀偏移函数，它们都是在为最后确定模式字符串的偏移量做准备。经过以上步骤就可以最终确定模式字符串的具体偏移量，最终的右移量是bmGs[i]和bmBc[b] $-$ m $+$ i $+$ 1两者中的大值。

这个算法的操作过程非常抽象，它的证明也要更为复杂，很多刚接触到它的人都不能很好地理解它。由于专门论述它的证明意义不大，但为了使读者进一步掌握这种算法的核心思想，下面举一个例子来说明它具体是如何应用的。

生物学研究表明，DNA（脱氧核糖核酸）是染色体的主要化学成分，同时也是组成基因的材料。DNA 是由核酸的单体聚合而成的聚合体，每一种核酸由 3 个部分所组成：一分子含氮碱基、一分子五碳糖和一分子磷酸根。核酸的含氮碱基又可分为 4 类：鸟嘌呤G、胸腺嘧啶T、腺嘌呤A和胞嘧啶C。因此含氮碱基序列是标识不同 DNA 之间的唯一区别。现在请问子序列GCAGAGAG是否存在于下面这段 DNA 碱基序列中。

GCATCGCAGAGAGTATACAGTACG

如图 3-15 所示，先将模式字符串与目标字符串左对齐，然后按照从右向左的方向进行比较。此时发现$P[i=7]$ = G与$T[i+j=7]$ = A不同，于是根据前面的方法，分别计算bmGs[7]和bmBc[A]。首先根据坏字符偏移函数来计算bmBc[A], 易知属于第 1 种情况。于是在模式字符串P中从右向左搜索字符A第 1 次出现的位置，即$P[i'=6]$处。所以有$i-i'=7-6=1$, 即得bmBc[A] $-$ 8 $+$ 7 $+$ 1 $=$ 1。再计算bmGs[7], 因为这里u为空，所以bmGs[7] $=$ 1。最终得到第 1 次的右偏移量为1。

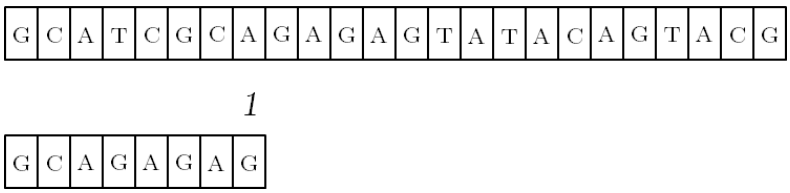


图 3-15　BM 算法示例 1

如图 3-16 所示，将模式串P向右移动1个单位。按照从右向左的方向进行比较，发现$P[i=5]$ = G与$T[i+j=6]$ = C不同，再分别计算bmGs[5]和bmBc[C]。先根据坏字符偏移函数来计算bmBc[C]，易知属于第 1 种情况。于是在模式字符串P中从右向左遍历搜索字符C第一次出现的位置，得到$P[i'=1]$，所以有$i-i'=7-1=6$，即得

bmBc[C] − 8 + 5 + 1 = 4。计算bmGs[5]，易知这里u = AG，模式字符串P中存在AG再次出现且前缀不为G的情况。将模式字符串最右端再次出现的且前缀不是G的片段与目标字符串中的 T[6]对齐，得到bmGs[5] = 4。最终得到第 2 次的右偏移量为4。

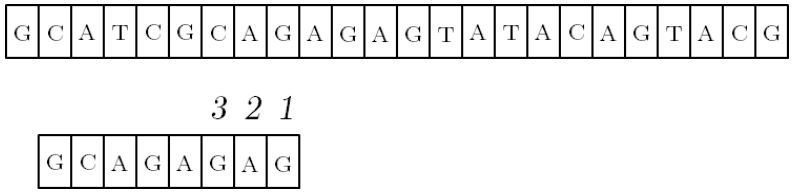


图 3-16　BM 算法示例 2

如图 3-17 所示，将模式字符串P向右移动4个单位。按照从右向左的方向将模式字符串与目标字符串进行比较，发现匹配成功。注意，匹配成功时不能再使用坏字符偏移函数或者好后缀偏移函数来确定右偏移量！因为它们是在发现不匹配时用来确定偏移量的。一旦匹配成功就将模式字符串直接右移$m - 1$个位置。以下的匹配过程与前面几步相似，这里不再赘言。后续过程如图 3-18 所示。当模式字符串移动到与目标字符串右对齐时，匹配结束。

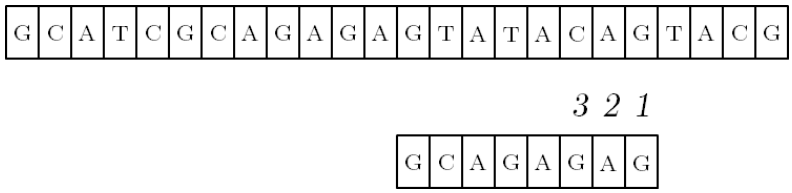


图 3-17　BM 算法示例 3

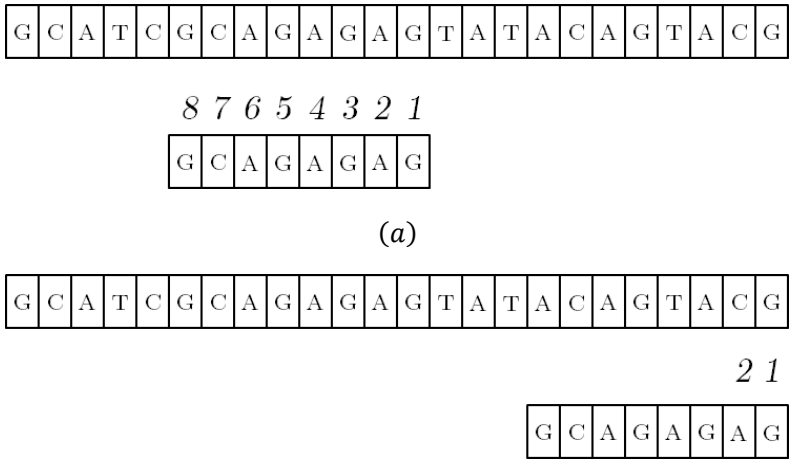


图 3-18　BM 算法示例 4

至此可知，BM 算法的关键是确定好后缀偏移函数和坏字符偏移函数。前面所讲的偏

移量确定方法仅是一个形式化的描述。下面给出关于好后缀偏移函数和坏字符偏移函数的一个标准化定义。

坏字符偏移函数是一个容量为σ的表，其中σ表示目标字符串中元素的种类数，例如上例中$\sigma = 4$，因为目标字符串中只含有A、T、G、C这 4 种元素。坏字符偏移函数的定义是：

$$\text{bmBc[char]} = \begin{cases} \min\{i: 1: \leqslant i < m-1, P[m-i-1] = \text{char}\}, & \text{char} \in P \\ m, & \text{char} \notin P \end{cases}$$

以上为例，模式字符串P可转化为表 3-7。

表 3-7　模式字符串P

i	0	1	2	3	4	5	6	7
$P[i]$	G	C	A	G	A	G	A	G

由此即可以通过上述定义构造出bmBc表，如表 3-8 所示。

表 3-8　bmBc表

char	A	C	G	T
bmBc[char]	1	6	2	8

好后缀偏移函数用来计算模式中的某个后缀被匹配成功时文本指针的右移偏移量，它由两个因素来决定。

- 该后缀在模式中除去自身外，从最右端开始再次出现的位置。
- 在第 1 个因素不成立的情况下，该后缀的某个后缀同时又是模式的前缀。

与坏字符偏移函数相同，好后缀偏移函数也是一个表，它的容量是m，即模式字符串P的大小。为了描述好后缀偏移函数，首先定义两个条件：

- 条件 1：对于每个k，使得$i < k < m$，$s \geqslant k$或者$P[k-s] = P[k]$；
- 条件 2：若$s < i$，则$P[i-s] \neq P[i]$。

条件 1 用于计算模式的某个后缀除去自身外的从最右侧开始重新出现时所对应的偏移量；条件 2 确保计算到模式的第i个位置。

好后缀偏移函数的定义可以表示为：

$$\text{bmGs}[i] = \begin{cases} m-1, & i = 0 \\ \min\{s > 0: \text{条件 1} \wedge \text{条件 2}\}, & 1 \leqslant i \leqslant m-1 \end{cases}$$

根据上述定义来填写bmGs表。首先很容易得到bmGs[0] = 7。除此项之外，其他各项均是$s > 0$情况下的最小值。观察条件1和条件2可以知道，$s < i$与$s \geqslant k > i$是不能同时成立的，所以最后要求的bmGs$[i]$分两种情况，其一是满足且仅满足条件“ 对于每个k，使得$i < k < m$，$s \geqslant k$ ”中的s的大于0时的最小值；其二是满足且仅满足条件 “ 对于每个k，使得$s < i < k < m$，$P[i-s] \neq P[i]$且$P[k-s] = P[k]$ ”中的s的大于0时的最

小值。

由于计算过程大多数相似，因此这里将不一一计算，仅选择bmGs[2]、bmGs[3]、bmGs[4]和bmGs[7]的计算过程进行说明。

计算bmGs[2], $i=2$, 根据条件$s<2$, 那么$s=1$。当$s=1$时, $P[i-s]=P[1]\neq P[2]$, 满足条件2。此时对于$i<k<m$，$k=\{3,4,5,6,7\}$，当$k=3$时，显然$P[3-1]\neq P[3]$，即$P[k-s]\neq P[k]$，继续验证下去，发现$k=\{3,4,5,6,7\}$时，$P[k-s]\neq P[k]$，所以这个条件判断失败。这时s应当满足$s\geqslant k$，即$s\geqslant\{3,4,5,6,7\}$，所以$s\geqslant 7$。bmGs[2]是s的最小值，所以bmGs[2] = 7。

计算bmGs[3], 此时$i=3$, 根据条件$s<3$, 那么$s=\{1,2\}$, $3<k<8$, 即$k=\{4,5,6,7\}$。当$s=1$时，$P[i-s]=P[2]\neq P[3]$；当$s=2$时，$P[i-s]=P[1]\neq P[3]$；所以对于$s=\{1,2\}$都满足条件 2。当$k=4$，$s=1$时，$P[3]\neq P[4]$，即$P[k-s]\neq P[k]$，所以有$s\neq 1$。当$k=4$且$s=2$时，$P[2]=P[4]$，即$P[k-s]=P[k]$。同理，继续验证下去，发现当$s=2$时，对于$k=\{4,5,6,7\}$中的每个值都满足$P[k-s]=P[k]$，所以$s=2$。再考虑另外一种情况，即当$s\geqslant k$的情形，易得$s\geqslant\{4,5,6,7\}$，所以$s\geqslant 7$。将两种情况合并起来，知s的最小值是2，所以bmGs[3] = 2。

计算bmGs[4], 此时$i=4$, 根据条件$s<4$, 则$s=\{1,2,3\}$, $4<k<8$, 即$k=\{5,6,7\}$。当$s=2$时，$P[i-s]=P[2]=P[4]$，不满足条件 2。这时s应当满足$s\geqslant k$，即$s\geqslant\{5,6,7\}$，所以有$s\geqslant 7$。bmGs[4]是s的最小值，所以bmGs[4] = 7。

计算bmGs[7]，此时$i=7$，根据条件$s<7$，那么$s=\{1,2,3,4,5,6\}$，k为空，显然无法计算。这时直接取$s>0$的最小值，也就是1，即bmGs[7] = 1。

综上可得如表 3-9 所示的结果。

表 3-9　bmGs表

i	0	1	2	3	4	5	6	7
$P[i]$	G	C	A	G	A	G	A	G
bmGs[i]	7	7	7	2	7	4	7	1

下面总结一下 BM 算法的核心思想并分析一下这样设计的好处。

首先，在 BM 算法中，模式字符串与目标字符串的匹配从右向左进行，这是因为考虑在匹配比较过程中不少的情形是前面的许多字符都匹配而最后的若干字符不匹配，结果导致匹配失败而又浪费了很多计算时间。

其次，BM 算法采用坏字符偏移算法对模式进行预处理以得到一些启发式信息，并使用这些信息来计算模式向右移动的距离。这个函数之所以称为“ 坏字符 ”就是因为它是针对模式字符串的某个位置的“ 坏字符 ”设计的。启发式规则是当发生匹配失败时，移动模式字符串使得失配的文本字符与该字符在模式字符串中的最右出现对齐。

再次，好后缀偏移函数是针对模式中某个子串的。当发生失配的情况时，如果已匹配

的子串长度不为0，就可以采用“好后缀”启发式规则，即移动模式字符串使得已匹配部分与该部分在模式字符串中的最右端出现对齐。

最后，取前面两个结果的较大者作为模式移动的距离。

通过前面的介绍，不难体会到 BM 算法的执行过程相当复杂。但由于该算法具有很高的执行效率，因而在实际中具有非常广泛的应用。从模式字符串右端开始向左进行比对的这种想法无疑是对传统思维方式的极大颠覆。更为重要的是，BM 算法还拓展了一块广阔的优化空间，以此为基础改进而来的算法形成了一个相当庞大的算法家族。

3.2.5　BMH 算法

算法

BMH 算法，它较 BM 算法要简单许多，但性能同样优异。

在 BM 算法的实际应用中，坏字符偏移函数的应用次数要远远超过好后缀偏移函数的应用次数，坏字符偏移函数在匹配过程中起到移动指针的主导作用。在实际匹配过程中，只使用坏字符偏移函数也非常有效。1980 年，奈杰尔 · 豪斯普（Nigel Horspool）提出了改进的 BM 算法——BMH 算法。算法将失配与计算右移量独立，计算右移量先判断$T[j+m-1]$是否等于$P[m-1]$，若相等则继续比较$T[j+m-2]$和$P[m-2]$，如此继续下去，直到$P[0]$等于$T[j]$，则发现一个匹配；若中间比较到$P[i]$和$T[j+i]$发现不相等，$T[i+j+1,\cdots,j+m-1]$已比较完毕且等于$P[i+1,\cdots,m-1]$（对应于u，其中c 是最后一个字符），则在$P[0,\cdots,m-2]$中看是否存在c字符，若不存在c字符，则模式直接向右移动m位。若发现$P[k]=c=T[j+m-1]$，其中$0\leqslant k\leqslant m-2$，则模式串右移$m-k-1$位，接着继续从右向左开始新一轮的比较。

BMH 既是对 BM 算法的改进，也是一种简化。BMH 算法的执行非常简便，而且效率也很可观。下面我们就用一个简单的例子来演示该算法的执行过程。假设有一个模式字符串P为 GTTAC，以及一个目标字符串T为 GCCTCATCCUACGTTAC。

首先，根据P来构建坏字符偏移表。本节开始已经给出了 BMH 算法中坏字符偏移表的构建方法。当发生失配时，字符$T[j+m-1]$不存在于P中时，右偏移量的值等于模式字符串的长度m。否则，如果字符$T[j+m-1]$在P中的位置索引为k的话，那么右偏移量的值等于$m-k-1$。所以构建坏字符偏移表时要从左向右逐个扫描模式字符串中的每个字符，对于那些重复出现的字符将以其最后出现的位置为准。对于最后一个字符，其偏移值的处理方法稍有不同。如果该字符在此前已经出现过，则保留其原值。否则就将字符串的长度作为该字符的偏移取值。据此便可得到本例中模式字符串的坏字符偏移表，如表 3-10 所示。

表 3-10　坏字符偏移表

字符	G	T	A	C	其他
偏移量	4	2	1	5	5

图 3-19 是一个应用 BMH 算法进行模式匹配的示例。每次匹配都从模式字符串的最右端开始。第 1 次失配的字符是 T，但是注意我们并不据此来计算偏移量，而是从目标字符串里已经完成匹配的部分子串中找到最右端的那个字符。在本例中也就是字符 C，

然后再根据坏字符偏移表，模式字符串右移 5 个位置。第 2 轮比较时，首字符即失配，且字符 U 也没有在模式字符串中出现过，所以将模式字符串整体右移 5 个位置。第 3 轮比较时，首字符 T 处发生失配，从坏字符偏移表中可以查到，T 对应的偏移量是 2，所以将模式字符串整体右移 2 个位置。最后一轮比较时，刚好完全匹配，任务结束。

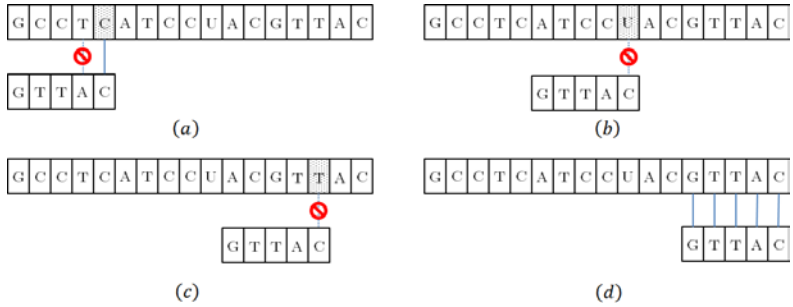


图 3-19　BMH 算法示例

下面给出 BMH 算法的实现源码。读者同样可以从本书的在线支持网站上下载得到用于验证下列函数使用的测试程序。

```
const int MAXCHAR = 256;

void PreProcess(const char * patt, int m, int bmBc[])
{
    int k = 0;
    for (k = 0; k < MAXCHAR; k++)
        bmBc[k] = m;

    for (k = 0; k < m - 1; k++)
        bmBc[patt[k]] = m - k - 1;
}

int BMS(string t, string p)
{
    int bmBc[MAXCHAR];
    int m = p.length();
    int n = t.length();
    if (m > n)
      return -1;

    const char * patt = p.c_str();
    const char * text = t.c_str();
    PreProcess(patt, m, bmBc);

    int k = m - 1;
    while (k < n) {
        int j = m - 1;
        int i = k;
        while (j >= 0 && text[i] == patt[j]) {
            j--;
            i--;
        }
        if (j == -1)
            return i + 1;
        k += bmBc[text[k]];
    }
```

```
    return -1;
}
```

以 BMH 算法为基础，又有学者在 1990 年左右提出了改进算法，即 BMHS 算法。该算法的主要改进思想是在计算坏字符偏移函数时，考虑下一个字符的情况，也即利用下一个字符决定右移量。当比较进行到文本串j位置时，失配发生在$T[i+j]=b$与$P[i]=a$处。只使用$T[j+m]=c$决定右移量，当下一个字符$T[j+m]$不在模式中出现时，它的右移量比 BMH 算法的右移量大，BMHS 算法右移$m+1$位，BMH 算法只能右移Badchar$[j+m-1]$位置，即便是$T[j+m-1]$不出现在模式字符串中，也最多移动m位。所以通常情况下，BMHS 算法比 BMH 算法快，但当$T[j+m-1]$没有在模式中出现，而$T[j+m]$却出现在模式中时，即当Badchar$[j+m-1]>$ Badchar $[j+m]$时，BMHS 算法的效果就不如 BMH 算法了。

3.3　文本的模糊匹配

在目标字符串中对模式字符串进行精确的匹配无疑是一种实际而普遍的需求。例如，若想把一篇文档中的“贞观元年”替换成“公元 627 年”，则此时所需用到的技术就是精确的模式匹配。但另外一种实际需求——模糊匹配，也大量存在。所谓模糊匹配，就是允许匹配结果与原始字符串之间存在一定程度的差异，而无须做到百分之百的一致。显然，模糊匹配是相对于精确的模式匹配而言的。

3.3.1　全局编辑距离

算法

德勒曼-温施算法，是编辑距离计算方面的经典算法，在生物信息学中具有重要应用。

主流的文字处理软件中一般都会提供有拼写检查功能。当用户输入一个错误的拼写，例如“baanana”，系统马上会提示原本要输入的字符串是否应该是“banana”。显然，从一个错误的拼写匹配到一个正确的拼写，文字处理软件使用的就是模糊匹配技术。此外，很多英文单词也都有不止一种拼写方式，比如“analyse”也可以写成“analyze”，“learned”也可以写成“learnt”。特别是一些非英语国家的人名被翻译成英文后，也常常会存在多种拼写形式。例如，苏联数学家格奥尔吉·阿杰尔松-韦利斯基的名字既可以写成“Georgy Adelson-Velsky”，也可以写成“Georgii Adelson-Velskii”。当你使用谷歌查询这些关键字的时候，搜索引擎返回的内容往往已经兼容了该词条的多种拼写形式。这种功能的实现同样依赖于模糊匹配技术。

从另外一个角度来说，对于一个搜索字符串，假使已经得到了多个可选的匹配结果，如何评判其中哪个才是最接近的呢？这时就需要用到编辑距离的概念。所谓编辑距离，是指两个字符串之间，由一个转换成另一个所需的最少编辑操作次数。许可的编辑操作包括将一个字符替换成另一个字符，插入一个字符以及删除一个字符。更小的编辑距离意味着一个更加接近的匹配。借由编辑距离，便可以对众多备选结果进行评估和打分，并最终实现对多个备选结果的排序。

1970 年前后，美国的两位研究人员索尔·德勒曼（Saul Needleman）和克里斯蒂安·温施（Christian Wunsch）共同提出了著名的德勒曼-温施（Needleman-Wunsch）算法。

该算法最初用于基因序列分析，现在它也是进行文本编辑距离分析的一种重要方法。此外，它还是历史上最早的应用动态规划思想设计的算法之一。

前面已经讲过，编辑操作包括将一个字符替换成另一个字符，插入一个字符以及删除一个字符。为了便于执行并易于核算，我们必须在具体化这些编辑操作的同时，提供一个评分方案。在德勒曼-温施算法中，字符间的关系被设定为以下几种类型。

- 匹配：两个字符是相同的。
- 不匹配：两个字符是不同的。
- 插入或者删除：一个字符对应另外一个字符串中的一个空缺。

对于最后一条，这里稍做解释。例如，有字符串ac和abc，显然若经由最少的编辑操作次数而将其中一个转换成另一个，则要么是从abc中删除b，要么是在ac中插入b。所以就认为字符b对应了另外一个字符串中的一个空缺。

可以为上面几种场景设计多种不同的评分方案。此处采用最简单的一种方案：令匹配的得分是$+1$，不匹配的得分是-1，插入或者删除的得分也是-1。为了便于读者理解，下面结合一个简单的例子来说明算法的具体执行过程。

算法开始，首先依据参与计算的两个字符串的长度来建立一个矩阵。假设第 1 个字符串的长度为M，第 2 个字符串的长度为N，则建立一个$M+2$行、$N+2$列的矩阵。并将第 1 个字符串中的字符逐个填入第 1 列中第 3 行开始直到第$M+2$行的每个位置中。同时，将第 2 个字符串中的字符逐个填入第 1 行中第 3 列开始直到第$N+2$列的每个位置中。至于为何是从第 3 个位置开始填写，原因在于我们认为字符串开始前，应该留有一个空缺位。其实你可以想象计算一个（非空）字符串和一个空字符串之间编辑距离的情况。因此所有参与算法执行的字符串都以一个空缺来作为开始。

例如，要进行编辑距离计算的两个字符串分别是 GATTACA 和 GCATGCA。第 1 个字符串的长度是7，第 2 个字符串的长度也是7，所以建立一个9×9的矩阵，并将 GATTACA 中的字符逐个填入第 1 列中第3行开始直到第9行的每个位置中。将 GCATGCA 中的字符逐个填入第 1 行中第3列开始直到第9列的每个位置中，如图 3-20 左图所示。

算法从矩阵的第2行第2列开始，并将该位置赋值为0。然后逐行逐列计算并填写剩余位置的值。方法是分别从上方、左方、左上方 3 个方向来计算该位置的值，并从中选择出最大值作为当前位置的值。假设当前要计算位于矩阵中$[i][j]$处的一个值，$[0][j]$处的字符用A表示，$[i][0]$处的字符用B表示，则有：

- 如果从$[i-1][j]$处（即左方一格）或者从$[i][j-1]$处（即上方一格）来计算$[i][j]$处的值，则表明在最后的对准过程中需要一个插入或者删除操作。产生备选值的方法就是将左方一格处和上方一格处的值分别减去1。因为前面规定插入或者删除的得分也是-1。

- 如果从$[i-1][j-1]$处（即左上方一格）来计算$[i][j]$处的值，则分两种情况。如果$A=B$，则产生备选值的方法就是将左上方一格处的值加1。因为前面规定匹配的得分是+1。如果$A\neq B$，则产生备选值的方法就是将左上方一格处的值减去1。因为前面规定不匹配的得分是−1。

显而易见的是，对于第 2 行的各个位置是没有所谓的上方一格和左上方一格的。所以这一行上的值仅能从左方一格算出。而且这种行为也有很好的解释，即对一个空字符而言，在最后的对准过程中只能采取删除或者插入操作。同理，对于第 2 列的各个位置是没有所谓的左方一格和左上方一格的。所以这一列上的值仅能从上方一格算出。于是便会得到如图 3-20 右图所示的结果。

		G	C	A	T	G	C	A
	0							
G								
A								
T								
T								
A								
C								
A								

		G	C	A	T	G	C	A
	0	-1	-2	-3	-4	-5	-6	-7
G	-1							
A	-2							
T	-3							
T	-4							
A	-5							
C	-6							
A	-7							

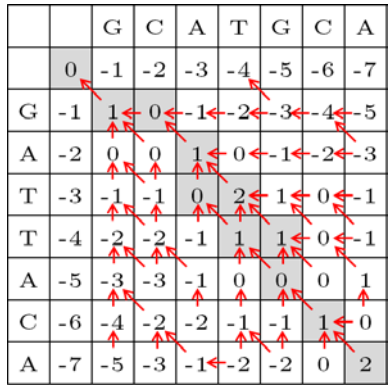
图 3-20　算法初始化操作

依据上面的规则，最终可以计算出矩阵中所有位置的值，如图 3-21 所示。矩阵中最右下角的值就是这两个字符串之间的编辑距离评分。在具体解释这个评分意义之前，我们先要从矩阵中得到一个最佳的编辑校准方案。因此，需要从最右下角的方格进行回溯，从而找到一条通向起始点（即第2行第2列的方格）的路径。图中的箭头表示了当前方格的值是由左方、上方、左上方中的哪个方格计算而来的。由这些箭头作为指引，就可以很容易地找到回溯路径。

还应该注意到，某些方格的值可以从不同的方向算得，例如第6行第6列处方格里的值既可以从上方得到，也可以从左上方得到。于是，最终得到的路径可能不止一条。来自左上方的计算意味着一次匹配或者不匹配，如果不匹配，在最后的处理时可以用一次替换操作来消除差异。来自上方或者左方的一次计算，意味着最终对准形式里的一次删除或插入操作。基于这个规则，最终可以得到如下 3 个最佳对准形式，这也对应了图 3-21 中用灰色方格连成的 3 条回溯路线。

```
GCATG_CA    GCA_TGCA    GCAT_GCA
G_ATTACA    G_ATTACA    G_ATTACA
```

无论是哪种对准方案，其中都包含有5次匹配，1次不匹配以及2次插入或删除操作。匹配操作对应+1分，其余操作对应−1分。所以最终编辑距离的评分就是2分。通过简单的分析也易知该算法的时间复杂度和空间复杂度都是$O(mn)$，其中m和n是两个字符串各自的长度。



		G	C	A	T	G	C	A
	0	-1	-2	-3	-4	-5	-6	-7
G	-1	1	0	-1	-2	-3	-4	-5
A	-2	0	0	1	0	-1	-2	-3
T	-3	-1	-1	0	2	1	0	-1
T	-4	-2	-2	-1	1	1	0	-1
A	-5	-3	-3	-1	0	0	0	1
C	-6	-4	-2	-2	-1	-1	1	0
A	-7	-5	-3	-1	-2	-2	0	2

图 3-21　德勒曼-温施算法示例

算法

史密斯-沃特曼算法，同样在生物信息学中具有重要应用。

3.3.2　局部最佳对准

另外一种寻找最佳编辑（对准）方式的算法是由美国的两位生物信息学家坦普尔 · 史密斯（Temple Smith）和迈克尔 · 沃特曼（Michael Waterman）于 1981 年共同提出的。该算法现在通常被称为史密斯-沃特曼（Smith-Waterman）算法。与试图从全局角度寻找最佳对准方式的德勒曼-温施算法不同，史密斯-沃特曼算法是一种局部序列对准算法。

史密斯-沃特曼算法的执行过程与德勒曼-温施算法非常相近，只是当计算结果出现负值时，以0代替。所以在对矩阵进行初始化时，第 2 行以及第 2 列上的值都将为0。假设有两个字符串 ACACACTA 和 AGCACACA，现在要寻找二者的最佳对准方式。图 3-22 给出了算法的执行矩阵结果，由于其中各值的计算方法与德勒曼-温施算法几乎无异（仅仅是在出现负值时以0代替），所以此处不再赘言。

		A	C	A	C	A	C	T	A
	0	0	0	0	0	0	0	0	0
A	0	2	1	2	1	2	1	0	2
G	0	1	1	1	1	1	1	0	1
C	0	0	3	2	3	2	3	2	1
A	0	2	2	5	4	5	4	3	4
C	0	1	4	4	7	6	7	6	5
A	0	2	3	6	6	9	8	7	8
C	0	1	4	5	8	8	11	10	9
A	0	2	3	6	7	10	10	10	12

图 3-22　史密斯-沃特曼算法示例

为了获得局部最佳对准，我们需要从矩阵中的最大值开始回溯。如果当前位置处在矩阵中的$[i][j]$处，那么向回倒退一个位置的话，则仅能从$[i-1][j]$、$[i][j-1]$和$[i-1][j-1]$中选择。此时，我们便需要根据之前构建该矩阵时的运动走向来进行选

择。路径回溯的过程将在到达一个0值点时终止。在给出的例子中，最大值位于矩阵的最右下角处，图 3-22 中已将回溯的路径用箭头标出。

一旦完成路径回溯，那么最佳对准的构建方法与德勒曼-温施算法类似：对角线方向上的一个移动表示一个对准（可能是匹配也可能是不匹配），一个上下方向上的移动表示一个删除操作，一个左右方向上的移动表示一个插入操作。据此，便可得到上例的最佳对准方式如下：

A_CACACTA
AGCACAC_A

史密斯-沃特曼算法同样是非常耗时的。假设要进行对准操作的两个字符串的长度分别为m和n，那么算法的时间复杂度即为$O(mn)$。在标准的执行过程下，史密斯-沃特曼算法的空间复杂度也为$O(mn)$。

3.3.3　*N* 元距离模型

算法

N-gram 算法，它是定义两个字符串间距离的另外一种方法。

我们除了可以定义两个字符串之间的编辑距离之外，还可以定义它们之间的N-gram距离。N-gram（有时也称为N元模型）是自然语言处理中一个非常重要的概念。假设有一个字符串s，那么该字符串的N-gram就表示按长度N切分原词得到的词段，也就是s中所有长度为N的子字符串。设想如果有两个字符串，然后分别求它们的N-gram，那么就可以从它们的共有子串的数量这个角度去定义两个字符串间的N-gram距离。但是仅仅是简单地对共有子串进行计数显然也存在不足，这种方案显然忽略了两个字符串长度差异可能导致的问题。比如字符串girl和girlfriend，二者所拥有的公共子串数量显然与girl和其自身所拥有的公共子串数量相等，但是我们并不能据此认为girl和girlfriend是两个等同的匹配。

为了解决该问题，有学者便提出以非重复的N-gram分词为基础来定义N-gram距离这一概念，可以用下面的公式来表述：

$$|G_N(s)| + |G_N(t)| - 2 \times |G_N(s) \cap G_N(t)|$$

此处，$G_N(s)$是字符串s的N-gram集合，N值一般取2或者3。以$N = 2$为例对字符Gorbachev和Gorbechyov进行分段，可得如下结果（我们用下画线标出了其中的公共子串）。

Go, or, rb, ba, ac, ch, he, ev

Go, or, rb, be, ec, ch, hy, yo, ov

结合上面的公式，即可算得两个字符串之间的距离是$8 + 9 - 2 \times 4 = 9$。显然，字符串之间的距离越小，它们就越接近。当两个字符串完全相等的时候，它们之间的距离就是0。

下面这个函数简单地实现了上述N-gram距离的计算功能。

```
int N_gram(string x, string y, int n) {

    int num_x = x.length()-n+1;
    int num_y = y.length()-n+1;
    int num_s = 0;

    for(int i = 0; i < num_x; i++){
        string sub_str = x.substr(i, n);
        if(y.find(sub_str)!=-1)
            num_s++;
    }

    return num_x + num_y - 2*num_s;
}
```

3.3.4 语音编码模型

对字符串进行模糊匹配的另外一种策略是利用英文字母的读音规则，将发音上相近的词汇作为模糊匹配的备选结果。从汉字输入的角度来说，如果用户使用的是拼音输入法，那么运用基于读音的模糊匹配技术对输入错误进行纠正是一个十分常用的策略。再比如前面曾经提到过的例子，“ Georgy ”也可以写成“ Georgii ”，“ Velsky ”也可以写成“ Velskii ”，显然如果运用发音规则对这些词进行模糊匹配将会得到非常理想的效果。基于语言的编码和模糊匹配也是自然语言处理中一个非常重要的话题。

算法

Soundex 算法，在数据库中应用广泛，克努特在他的巨著《计算机程序设计艺术》中也有提及。

在基于语言的编码与匹配中，最重要也最普遍被采用的算法有两种，即 Soundex 算法和 Phonix 算法。其中 Soundex 算法是最早被提出的语音算法之一，现代语音算法基本都以此为基础改进而来。克努特在他的巨著《计算机程序设计艺术》的第 3 卷也曾提及过此算法。正如前面所提到的，作为一种语音算法，Soundex 根据英语发音的规则为字符串建立编码，最终的编码值由 4 个字符构成，第 1 个字符为英文字母，后 3 个为数字。其目的就在于将读音相近的（尽管在拼写上不完全一致）词汇编码成一致的表达式。

现今，Soundex 算法的应用非常广泛，包括 DB2、PostgreSQL、MySQL 和 Oracle 等在内的主流数据库系统中都有用到该算法。下面就具体来介绍该算法的执行过程。首先，保留字符串中的第 1 个字母，将其余字母按照表 3-11 所示的编码方案进行替换。然后，删除相邻的重复编码（该过程主要是为了剔除编码中的元音成分）。再删除所有为0的编码。最后返回结果字符串中的头 4 个字符，其中第 1 个字符是原字符串的首字母，后 3 个字符是 3 个数字，取值范围就在1~6这 6 个数字中。若最后的编码不足位，则在结尾处用0补齐。

表 3-11　Soundex 编码表

编　　码	字　　母
0	a,e,i,o,u,h,w,y
1	b,f,p,v
2	c,g,j,k,q,s,x,z
3	d,t

续表

编　　码	字　　母
4	l
5	m,n
6	r

例如，根据上述规则，很容易得出字符串Jones和Janes的 Soundex 编码都是 J520。不仅如此，基于一些简单的规则，我们还可以根据相似程度对被选方案进行分级。例如，现在需要在下面几个单词中：

harriet, friend, mouse, hurry, humored

找寻与harriett相符合的模糊匹配。其对应的 Soundex 编码为 H630。其他字符串的编码分别为 H630、F653、M200、H600 和 H563。据此，我们可以认为harriet显然是一个最好的匹配，hurry和humored可以认为是能够被接受的匹配，而friend和mouse则是两个糟糕的匹配。

当然，由于 Soundex 算法的编码具有一定的局限性，例如码长有限等，它也会错将一些发音不同的字符串赋予相同的编码。例如，Catherine和Cotroneo的读音就完全不同，但是它们却具有相同的 Soundex 编码。

另外一种被常常用到的语音编码算法是 Phonix 算法。尽管它的执行过程与 Soundex 算法十分相像（这也导致很多人常常将两种算法混淆），但二者之间仍然存在区别。这种区别主要体现在它所执行的编码不同，如表 3-12 所示。显然这种编码方案相对于 Soundex 编码会得出更加细化的结果，这也易于区分一些被 Soundex 算法误判的编码。

算法

Phonix 算法，是对 Soundex 算法的改进，二者仅仅是编码表有所不同。

表 3-12　Phonix 编码表

编　　码	字　　母
0	a,e,i,o,u,h,w,y
1	b,p
2	c,g,j,k,q
3	d,t
4	l
5	m,n
6	r
7	f,v
8	s.x,z

本章参考文献

[1]　Saul B. Needleman, Christian D. Wunsch. A General Method Applicable to the Search for Similarities in the Amino Acid Sequence of Two Proteins. Journal of Molecular Biology, Vol. 48, No. 3: 443~453, 1970.

[2] James H. Morris, Vaughan R. Pratt. A Linear Pattern-matching Algorithm. Technical Report 40, University of California, Berkeley, 1970.

[3] Donald E. Knuth, James H. Morris, Vaughan R. Pratt. Fast Pattern Matching in Strings. SIAM Journal on Computing, Vol. 6, No. 2: 323~350, 1977.

[4] Robert S. Boyer, J Strother Moore. A Fast String Searching Algorithm. Communications of the ACM. Vol. 20, No. 10: 762~772, 1977.

[5] R. Nigel Horspool. Practical Fast Searching in Strings. Software - Practice and Experience. Vol. 10, No. 6: 501~506, 1980.

[6] Temple F. Smith, Michael S. Waterman. Identification of Common Molecular Subsequences. Journal of Molecular Biology, Vol. 147, No. 1: 195~197, 1981.

[7] Justin Zobel, Philip Dart. Phonetic String Matching: Lessons from Information Retrieval. Proceedings of the 18th International ACM SIGIR Conference on Research and Development in Information Retrieval. Zürich, Switzerland: 166~173, 1996.

[8] 张娜，张剑．一个快速的字符串模式匹配改进算法．微电子学与计算机，2007年，第 24 卷，第 4 期.

第 4 章

链表——老鹰捉小鸡

数组可以被认为是一种最简单、最朴素的数据结构。用数组的形式可以实现对数据的“串接”。即先在内存中开辟一块连续的区域，然后把数据依次存储进去。当访问其中的一个或几个数据时，可以利用这些数据的索引进行操作。这样的结构虽然实现简单，操作方便，但也致使程序的可伸缩性大大降低。数组是定长的静态的数据结构。而现实中的很多问题都是随着客观情况的变化而动态改变的。因此使用一个静态的数组来处理问题总是会遇到一些这样或那样的问题。为了解决这种矛盾，人们引入了动态数据结构的概念。动态数据结构就是在程序执行过程中能够动态地扩展或收缩的一种数据组织形式。链表就是一种最简单的动态数据结构。

4.1 链表的概念

小提示

链表是实现许多数据结构的基础，链表的使用同时也考察了读者对于 C++中指针和动态内存管理的掌握情况，因此熟练掌握它是参加 IT 公司笔试的必要准备。

前面我们提到，数组是一种定长的静态数据结构，缺乏伸缩性，因而面对很多不能事先确定的任务时，局限性很大。例如，现在需要将一幅数字图像的像素矩阵存入数组，那么在不知道该图像的具体信息之前通常这个数组需要被定义得非常大以便能够容下任意大小的图像数据。然而当图像较小时，显然会造成内存的浪费，并且一旦程序载入一幅足以超出数组容量的大图片时，那么就可能导致程序的崩溃。

为了避免出现上面这种尴尬，提高程序的可伸缩性，人们引入了链表这种数据结构。它不但能够存储较大的数据，还能够动态地调整容量。所谓链表，顾名思义，是种把数据像链条一样“串接”起来的数据结构。结构中各个数据就像链条上的环节，通过一定的手段组织这些环节，最后便会形成一条规整的数据链。

数据在计算机中的存储就像货物在仓库中的存储一样，不但占用一定的空间，还要有一个标识存储位置的地址。计算机通过该地址找到这个数据，然后再进行相关操作。链表的实现便利用了数据的地址。链表中的每一个“节点”除了存储数据外，还存储指向下一个节点的指针。于是，当计算机完成了对某个数据的访问后，便可以依据当前节点中存储的地址信息，找到下一个节点，进而完成对下一个节点中数据的访问和操作。鉴于链表的这种链式存储结构，若需要增加数据，就只需把数据链入链条，当需要减少数据时，只需把数据从链条中摘除。程序员只要进行增加或删除数据的操作，而无须过多关心程序在内存分配方面的问题。这大大增强了程序的可伸缩性，而且维护代价也降低了很多。

总的来说，链表是一种常见的线性数据结构。它被定义为一个节点序列，其中除了最后一个节点外的每个节点都包含有下一个节点的地址。整个序列就好像一串珠链一样，呈“线性”排列。实际中的链表有很多种。本章将重点讲解 3 种链表：单向链表、单向循环链表和双向循环链表。

4.2 单向链表

单向链表是所有链表中最简单的一种，之所以称之为单向链表是因为只能从一个方向上对其进行遍历，而且遍历不能循环进行。本节就以其中最简单的单向链表为例来向读者介绍有关链表的一些基本知识，注意如果没有特别说明，文中出现的链表就是指单向链表。

4.2.1 单向链表的结构

在单向链表中，元素的存放呈现“线性”结构，利用单向链表类和它提供的方法可以创建、删除以及修改一串“线性”的数据。单向链表与数组在很多方面都有相似之处，例如这两种数据结构都支持对元素的删除、遍历以及其他的一些基本操作。但数组与单向链表在完成一些特殊操作的效率上却存在很大差别。例如，在单向链表中可以很方便地移动第 1 个元素，但在数组中却不可能做到这样。另一方面，数组可以对元素进行随机访问，但单向链表却不可以。这些差别是由数组与单向链表在内存中存储方式的不同造成的。有关单向链表在内存中的存储方式将在稍后介绍。

问题导入

想想在幼儿园玩过的“老鹰捉小鸡”游戏。

现实中一个能够比拟成单向链表的例子就是“老鹰捉小鸡”。这个游戏相信大家一定都玩过。如图 4-1 所示，在这个游戏中，除了一个孩子扮演老鹰，一个孩子扮演母鸡以外，其他孩子都是小鸡。来看看母鸡和小鸡之间的连接关系：母鸡冲在队伍的最前面，然后有一只小鸡紧紧拉住母鸡，再后面的小鸡将依次拉住前面的小鸡，从而形成一个连续的线性结构。

图 4-1 老鹰捉小鸡

链表亦是如此，链表中的每个元素被称为“节点”。每个节点都包括两个部分，即实际数据和下一个节点的地址。每个链表都有一个表头指针（head）和一个表尾指针（tail）。表头指针指向表头节点，表头节点相当于老鹰捉小鸡游戏中的母鸡，相对应地，表尾

指针指向表尾节点，表尾节点相当于老鹰捉小鸡游戏中位于队尾的那只小鸡。

具体实现时，对于表头节点的处理方式可以有多种。我们可以认为它就是普通节点中的一员，也就是说，其中既存储实际数据也存储下一个节点的地址。另外一种处理方法是将表头节点看作是单向链表中第一个节点的前驱，即其中不包含实际数据而仅存放一个地址。该地址指向一个元素，后一种处理方式可以便于统一对空表与非空表的操作，进而使程序实现变得简单。

表尾指针指向单向链表的最后一个节点，这个节点不再指向其他节点，它称为表尾，存储的指针值为“ NULL ”。如果表头节点中的指针为“ NULL ”，那就表明此刻表头节点同时也是表尾节点，即该链表是一个空链表。

表头指针标识了单向链表的开始，表尾指针标识了单向链表的结束。由此可见，要找到某一个单向链表，只需找到它的表头指针即可。要找到某一个节点，则必须先找到它的前驱节点，然后再由前驱节点存储的指针找到该节点。在遍历链表时，如果某一个节点存储的指针值为“ NULL ”，则这个节点为表尾，单向链表到此结束。单向链表的标准结构如图 4-2 所示。

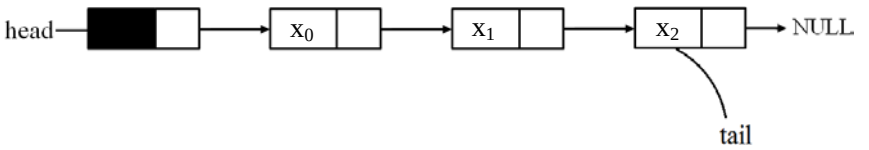


图 4-2　单向链表的结构

单向链表在内存中的存放并不像如图 4-2 所示的那样是“ 连续 ”的。实际上单向链表的各个节点分散存储在内存中的不同区域，然后再通过指针把这些节点“ 连接 ”起来。单向链表在内存中的实际存储形式如图 4-3 所示。当对一个单向链表进行遍历时，先找到它的表头节点，然后根据表头节点中存储的指针找到下一个节点，如此循环下去直到遍历完成。

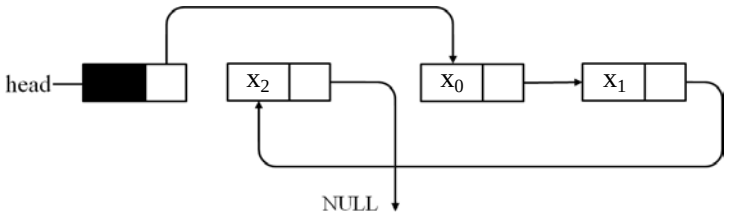


图 4-3　链表的实际存储

链表在内存中非连续的存储方式有优点也有不足。这种存储方式有利于节点的删除与插入，因为只要简单地更改几个指针就可以了。比如，用这种方法可以很容易地删除如图 4-3 所示链表中的第 2 个节点。具体过程就是令原来指向第 2 个节点的指针指向第 3 个节点。但是另一方面，这种存储方式不利于对链表中的节点进行随机访问。为了访问链表中某一个指定节点，必须从链表头开始遍历整个链表直到找到要访问的节

点为止。链表不连续的存储方式使程序不能仅依靠计算得到节点的地址，这与数组是截然不同的。数组利用内存中连续的区域存储数据，其中任何数据的地址都可以根据数组的首地址计算获得。

除了标识单向链表的开始与结束以外，表头指针与表尾指针还可用于不同单向链表间的合并。当要合并两个单向链表时，只需要根据表尾指针找到一个单向链表的表尾，再使这个表尾存储的值为“ NULL ”的指针指向另一个单向链表的表头即可，整个过程如图 4-4 所示。

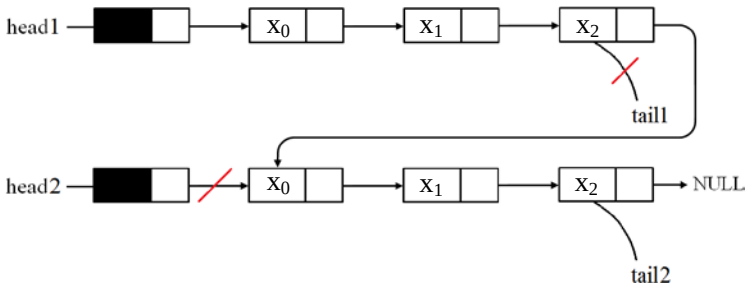


图 4-4　单向链表间的合并

4.2.2　单向链表的操作算法

一个类所完成的功能是由这个类的接口规定的。单向链表类的接口规定了单向链表所完成的功能。单向链表中的每一个节点不仅存储数据，还要存储指针，而且程序应该能对这个指针进行操作。因此，节点还应该封装对指针进行操作的方法。当单向链表类需要修改节点存储的指针时，直接调用这些方法。基于这些要求，节点也应该是一个类，而不是单纯的数据的结合。在介绍单向链表类的接口之前，先介绍节点类的接口。

求职面经点拨

请在完成本部分内容学习后，试回答附录题库中的第 21 题。

1．节点类与链表类

节点类封装的基本方法有：获得节点存储的指针以及修改节点存储的指针。节点类中还可以封装更多、更复杂的方法，但一个单向链表的实现除了要利用节点类外，还要用到单向链表类。如果节点类中封装了太多的方法，会使得节点的存储开销增加，最终导致整个单向链表的开销增加。当链入链表的数据量很大时，这种开销甚至可能是巨大的，因此应尽量把节点类设计得精简些。

现给出 ListNode 类的定义如下。

```
template <class T> class ListNode{

    T data;
    ListNode<T> * link;

public:

    ListNode() : link(NULL){}
    ListNode(T value) : link(NULL), data(value){}
```

```
    ~ListNode(){};
    void SetLink(ListNode<T> * next);
    void SetData(T value);
    ListNode<T> * GetLink();
    T& GetData();
};
```

以上程序使用了模板机制，这会使最终的链表类变得更加灵活、更加方便，而且在获得方便的同时，类的存储开销并未有明显变化。其中，函数 ListNode()为默认的构造函数，函数 ListNode(T value)为带一个参数的构造函数，它们的作用是初始化一个节点。当新建一个节点时，由于它还未链入单向链表，因此它存储的指针指向“ NULL ”。新建节点存储的数据可以由系统决定，也可以由参数 value 指定。

函数 SetLink(ListNode<T>* next)把参数 next 的值赋给节点中的数据成员 link；函数 GetLink()、GetData()分别返回节点的数据成员 link 及 data。下面给出这些函数的实现代码。

```
template <class T>
void ListNode<T>::SetLink(ListNode<T>* next){
    link = next;
}

template <class T>
void ListNode<T>::SetData(T value){
    data = value;
}

template <class T>
ListNode<T>* ListNode<T>::GetLink(){
    return link;
}

template <class T>
T&  ListNode<T>::GetData(){//返回节点中的数据
    return data;
}
```

以上函数的实现过程比较简单。这里需要说明的一个问题是，如果节点为 int 型，在调用 ListNode(T value)执行构造时就要给构造函数传递 int 型的参数。

在实现了节点类后，下面介绍单向链表类 List。List 类要封装的数据成员有：表头指针 head 以及表尾指针 tail，这两个指针标识了单向链表的开始与结束。正如前文所述，该类为每一个单向链表添加一个表头节点，表头节点存储的指针指向链表第 1 个实际存储数据的节点。由此可见，即使一个单向链表中什么数据也不存，它至少还拥有一个表头节点。而此时表头指针与表尾指针均指向这个节点，如图 4-5 所示。

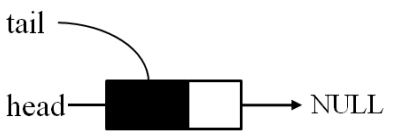


图 4-5　空单向链表

下面给出 List 类的定义。

```
template <class T> class List{

ListNode< T >* head;
ListNode< T >* tail;

public :

     List();
    ~List();

    bool AddTail (T value);
    bool RemoveTail ();
    bool InsertAt (int index , T value);
    bool RemoveAt (int index);

    T& GetAt (int index);
    bool IsEmpty ();
    int GetCount ();
    void RemoveAll ();

    ListNode< T >* GetHead();
    ListNode< T >* GetTail();
    ListNode< T >* GetNodeAt(int index);
    ListNode< T >* GetCur();
    ListNode< T >* TowardCur();
};
```

函数 AddTail(T value)用来向单向链表的表尾添加节点。函数 RemoveTail()用来删除表尾节点。函数 InsertAt(int index,T value)用来在索引 index 指定的位置添加一个数据成员是 value 的节点。函数 RemoveAt (int index)用来删除指定位置的节点。函数 GetAt (int index)根据索引 index 返回单向链表中相应节点所保存的 data 值。函数 IsEmpty()用来判断单向链表是否为空。函数 GetCount()用来获得当前链表中节点的个数（表头节点除外）。函数 RemoveAll () 用来删除链表中所有的节点。索引 index 即是链表中实际保存数据的节点的序号。规定距离表头节点最近的节点的 index 值为 0。

函数 GetHead ()、GetTail ()和 GetNodeAt (int index)分别返回链表表头指针、表尾指针以及指向其他节点的指针（这个节点的索引值为 index）。这几个函数为链表之间的连接提供了很大方便。

函数 GetCur()用来返回 cur。函数 TowardCur()使指针 cur 向后移动一个节点，并返回移动后的 cur。

2．插入操作的实现

在向单向链表的表尾插入新节点时，首先把表尾的 NULL 指针指向新加节点，这可以用函数 SetLink(ListNode<T>* next)完成，然后再把指向原链表的表尾指针向后移动一个节点即可，如图 4-6 所示。

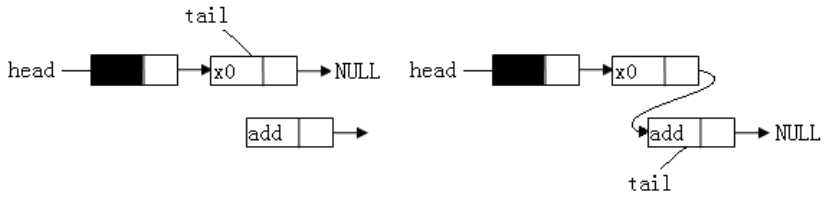


图 4-6　在表尾插入节点

向表尾加入新节点的成员函数实现如下。

```
template <class T>
bool List<T> :: AddTail (T value){
    ListNode<T>* add = new ListNode<T> (value);
    tail->SetLink (add);           //把新建节点链入表尾，成为新表尾
    tail = tail->GetLink();        //移动表尾指针置新表尾处
    tail->SetLink (NULL);
    if (tail != NULL)
        return true;
    else
        return false;
}
```

插入一个新节点，且该新节点将作为表头节点来使用，这时只需要让表头指针指向这个新节点，再让该新节点的指针指向原表头节点即可，整个过程如图 4-7 所示。

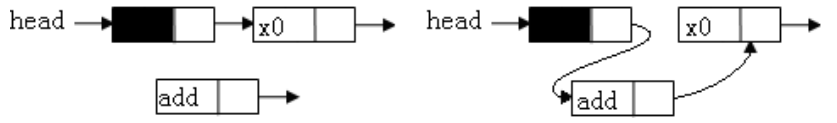


图 4-7　在表头位置插入节点

在向链表中其他位置插入新节点时，表尾 tail 指针不变，但要改变链表中相应节点存储的指针，改动过程如图 4-8 所示。可见插入位置的前一个节点的指针将指向新增节点，而新节点的指针则指向插入位置的后一个节点。

表头节点的引入使上述两种情况得以统一，由此，单向链表的插入操作变得很方便：向任何位置插入新节点时，表头指针 head 保持不变，而只需变动节点之间的指针即可。对于空链表的情况或者新插入的节点将作为表头节点使用时，只需变动表头指针指向这个新表头即可。

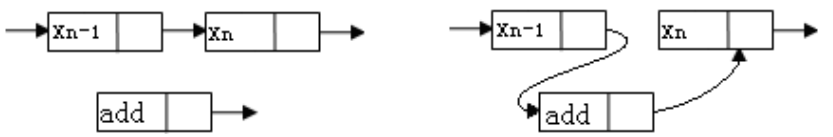


图 4-8　在其他位置插入节点

向在索引值指向的节点前插入一个新节点的成员函数实现如下。

```
template <class T >
bool List< T >::InsertAt ( int index,T value ) {
    //判断索引值是否正确
```

```
    if(index > this->GetCount ()-1 || index<0 ) {
        cout<<"A wrong position!\n";
        return false;
    }

    ListNode<T>* current = head;        //从头开始寻找出入地点

    while(index) {
       current = current->GetLink ();  //如果没找到，则顺序移动 cur
         --index;
    }

    ListNode<T>* add = new ListNode<T> (value);
    add->SetLink (current->GetLink ());          //将新建节点链入链表
    current->SetLink (add);

    if (current->GetLink () != NULL)
        return true;
    else
        return false;
}
```

3．删除操作的实现

删除节点的过程与上述插入节点的过程相反，举例来说，如图 4-9 所示是一辆列车，可以看出它由许多个车厢共同组成。除最后一节车厢外，每节车厢都连在前一个车厢后面，每节车厢后面又都连着一节车厢。现在如果车厢 B 有了故障，需将其从这辆列车中删掉，那么实际的做法是怎样的呢？简单地说，就是让车厢 A 后面不再与车厢 B 相连，而是与车厢 C 相连，也就是解除车厢 B 与车厢 A 和 C 之间的连接关系。

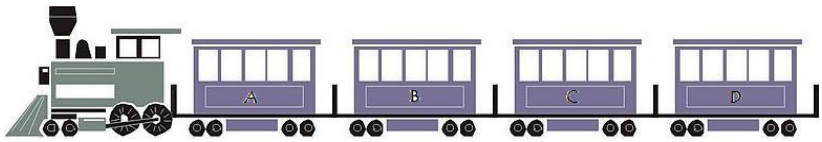


图 4-9　火车车厢

在链表中维系节点间链接关系的关键在于每个节点中指向下一个节点的指针，如果要删除一个节点 N，其实就是让 N 的前驱节点的指针不再指向 N，而是改为指向 N 节点的后继节点。如图 4-10 所示，图中的左图表示删除表尾节点，右图表示删除链表“中间”的某个节点。

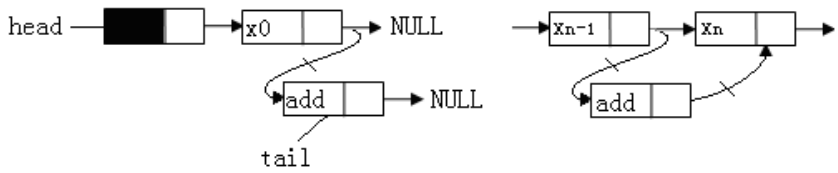


图 4-10　删除节点

删除操作的实现代码如下。

```
template <class T>
bool List<T>::RemoveTail() {//删除表尾节点
    //利用 RemoveAt（int index）实现
    return RemoveAt(this->GetCount()-1);
}

template <class T>
bool List<T>::RemoveAt(int index){ //按照索引值删除节点
    if(index > this->GetCount ()-1 || index<0){
        cout<<"A wrong position!\n";
        return false;
    }

    //用两个节点指针协作完成删除。其中，cur 指向要删除节点的前一个节点
    //preCur 指向要删除的节点
    ListNode<T>* cur,* curPre;
    cur = head;                  //从链表头开始寻找 index 指向的节点
    curPre = cur->GetLink();    //置 preCur 到 cur 后

    while(index){
        cur = cur->GetLink();   //如果没找到，则 cur 与 preCur 继续后移
        curPre = curPre->GetLink();
        --index;
    }

   //如果要删除的节点位于链表尾，则把 cur 置为新的表尾
    if(tail == curPre)
        tail = cur;
    cur->SetLink (curPre->GetLink());  //将被删除节点从链表中摘下
    delete curPre;

    if(curPre != NULL)
        return true;
    else
        return false;
}
```

4．其他操作的实现

下面给出 List 类中其他主要成员函数的实现代码，限于篇幅我们略去了部分非核心函数的实现代码，读者可以从本书的在线支持资源中下载得到完整的源码文件。

```
template <class T>
List<T>::List () {//创建表头节点，并初始化 head 与 tail
   head = new ListNode<T>();
   tail = head;
   tail->SetLink (NULL);
}

template<class T>
List<T>::~List()
{
   RemoveAll();
   delete head;
}

template <class T>
T&  List<T>::GetAt(int index) {    //返回索引值节点中的 value
```

```
    if (index > this->GetCount()-1 || index<0) {
        cout<<"A wrong position!\n";
    }

    ListNode<T>* cur;
    cur = head->GetLink();
    while(index){          //如果没找到，cur 顺序后移
        cur = cur->GetLink();
        --index;
    }
    return cur->GetData (); //返回节点中的 value
}

template <class T>
bool List< T >::IsEmpty () { //判空函数
    return head ->GetLink()==NULL ;
}

template <class T>
int List< T >::GetCount() {        //返回链表中节点的个数（表头节点除外）
    int count = 0;                 //用 count 记录节点个数
    //初始化指针指向第一个节点（不是表头节点）
    ListNode<T>* current = head->GetLink ();
    while(current != NULL) {
        ++count;
        current = current->GetLink ();  //顺序移动 cur
    }
    return count;
}

template <class T>
void List< T >::RemoveAll() { //删除所有节点
    ListNode<T>* cur;
   //当表头节点后还有其他节点时，删除这些节点
    while(head->GetLink() != NULL){
        cur = head->GetLink();
        head->SetLink (cur->GetLink());
        delete cur;
    }
    tail = head; //置表尾为表头处
}

//返回指向索引值为 index 的节点的指针
template <class T>
ListNode< T >* List< T >::GetNodeAt(int index){

    //判断索引值是否正确
    if(index > this->GetCount()-1 || index<0){
        cout<<"A wrong position!\n";
    }
    //handle 即为所求指针，它指向索引值是 index 的节点。
    ListNode< T >* handle = head->GetLink();
    while(index){ //寻找索引值为 index 的节点
        handle = handle->GetLink();
        --index;
    }
    return handle;
}
```

有了以上单向链表类后就可以对单向链表进行各种操作了。例如，对数据进行排序，合并两个链表，计算数据的平均值等。

4.2.3　有序链表的合并算法

有两个单向链表，其中每个链表都按照从小到大的顺序存储着一些整数。现在要求将两个链表合并成一个，并且合并后链表中数据的次序不变。

分析：如何实现对两个链表中数据的“融合”？可以考虑采用如下方法解决，每次取出 listSecond 中的第 1 个节点，把这个节点插入到 listFirst 中相应的位置。当插入完毕后删除这个节点，然后进行下一轮插值。如此循环下去，直到 listSecond 为空。

核心代码如下。

```
//创建两个链表 listFirst、listSecond
List<int> listFirst;
List<int> listSecond;

//初始化链表 listFirst
listFirst.AddTail(1);
listFirst.AddTail(6);
listFirst.AddTail(8);
listFirst.AddTail(9);
listFirst.AddTail(13);

//初始化链表 listSecond
listSecond.AddTail(0);
listSecond.AddTail(3);
listSecond.AddTail(4);
listSecond.AddTail(6);
listSecond.AddTail(11);
listSecond.AddTail(17);

while(listSecond.GetCount() != 0){ //当 listSecond 非空时持续循环
int indexFirst = 0;
//每次把 listSecond 中的第 1 个数按序插入到 listFirst 中
//用 while 循环语句寻找插入位置
    while(listSecond.GetAt(0)>listFirst.GetAt(indexFirst)){
        ++indexFirst;
      //如果 listFirst 已到链表尾，结束本次循环
        if(indexFirst = =  listFirst.GetCount()){
                break;
        }
    }
    if(indexFirst = = listFirst.GetCount()){//插入位置在 listFirst 链表尾
        listFirst.AddTail(listSecond.GetAt(0));
        listSecond.RemoveAt(0);
    }
    else{//插入位置在 listFirst 链表中
        listFirst.InsertAt(indexFirst,listSecond.GetAt(0));
        listSecond.RemoveAt(0);
    }
}
```

4.3 单向循环链表

在单向链表中，从一个已知节点出发，只能访问该节点及其后续的节点，而无法找到该节点之前的节点。为了解决这个问题，单向循环链表应运而生。在单向循环链表中，从任意节点出发都可访问链表中的所有节点，这一优点使某些运算在单向循环链表上易于实现。

4.3.1 单向循环链表的结构

求职面经点拨

请试回答附录题库中的第 22 题。

单向循环链表与单向链表在结构上极为相似。不同的是，单向循环链表的表尾节点存储的指针不为“ NULL ”，而是指向该链表表头节点的指针。由此可见，单向循环链表是一种首尾相接的链表。单向循环链表的结构如图 4-11 所示。

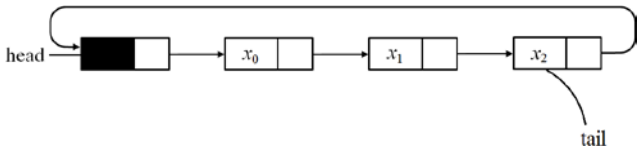


图 4-11 单向循环链表结构

与单向链表一样，单向循环链表也有表头节点、表头指针和表尾指针。表头指针与表尾指针为实现不同单向循环链表之间的连接提供了方便。连接过程如图 4-12 所示，即首先将第 1 个链表的表尾指针指向第 2 个链表的第 1 个节点，注意不是表头节点。然后，将第 2 个链表的表尾指针指向第 1 个链表的表头节点，并删除第 2 个链表的表头节点。从中易见，单向循环链表间的连接操作要比单向链表间的连接操作复杂得多。

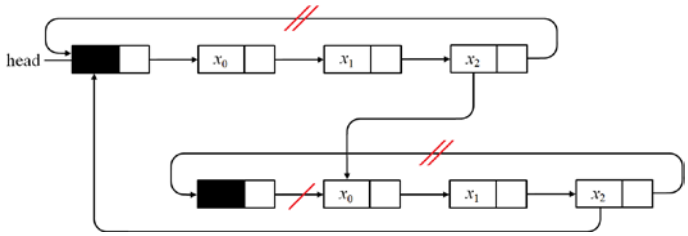


图 4-12 单向循环链表的合并

单向循环链表还有另一种实现形式。由于在单向循环链表中，表尾指针指向表头，所以表头指针可以用表尾指针代替。这种简化的单向循环链表中只有表尾指针而无表头指针，其实现形式如图 4-13 所示。

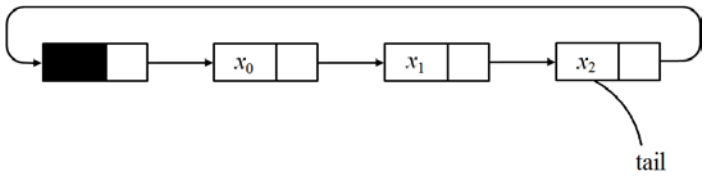


图 4-13 单向循环链表的另一种结构

由单向循环链表的方向性可知，表尾指针找到表头指针的时间复杂度为$O(1)$，而表头指针找到表尾指针的时间复杂度为$O(n)$。可见，只存储表尾指针是方便可行的。

与单向链表一样，单向循环链表在内存中的存储也不是连续的。各个节点分散地存储在内存中，最后再通过指针连接起来。图 4-14 表示了单向循环链表在内存中的存储形式。

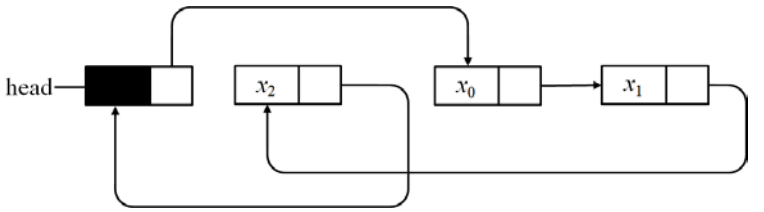


图 4-14　单向循环链表在内存中的存储形式

4.3.2　单向循环链表的实现

由于单向循环链表与单向链表仅仅是表尾节点存储的指针不同，所以用来实现单向链表的节点类 ListNode 同样可以用来实现单向循环链表。具体内容，读者可以参阅前面的类 ListNode，这里不再赘述。

下面介绍单向循环链表类（CirList）。

类 CirList 中要封装的数据成员有表头指针（head）、表尾指针（tail），由于可用表尾指针代替表头指针，所以类 CirList 中只封装 tail 即可。但为了叙述方便，仍然封装表头指针 head。

同单向链表一样，这里仍然使用一个表头节点。该表头节点不存储实际有效的数据。一个单向循环链表即使为空，也至少要保留一个表头节点。与单向链表不同的是，空的单向循环链表的表头节点中存储的指针仍然指向这个表头节点，如图 4-15 所示。

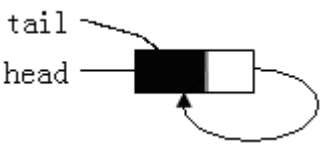


图 4-15　空单向循环链表

在向单向循环链表中插入新节点时，若新节点插入的位置不在表尾，则插入的方法与向单向链表中插入新节点的方法相同；若新节点插入的位置在表尾，则需要更改表尾指针，具体过程如图 4-16 所示。

同样地，在删除单向循环链表中的一个节点时，如果这个节点不在表尾，则删除的过程与从单向链表中删除一个节点的过程相同；如果这个节点在表尾，则需改动表尾指针，具体过程如图 4-17 所示。

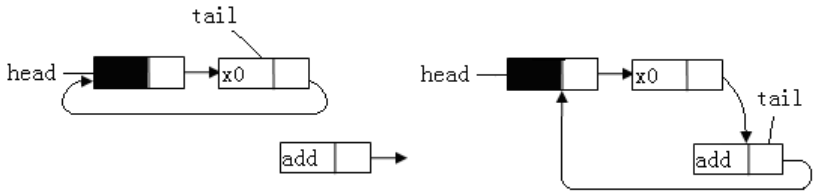


图 4-16　在表尾插入新节点

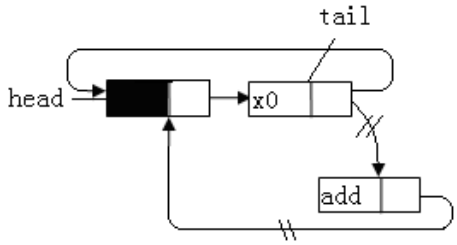


图 4-17　删除表尾节点

单向循环链表的判空也比较简单，只需要判断表头节点存储的指针是否仍指向表头节点即可。

在遍历单向循环链表时，由于链表在形式上呈现出闭合的环状，所以有必要再设置一个指针（cur），这个指针指向当前正被访问的节点。这样做是为了防止对链表进行无限循环的访问，同时也清晰地标记出当前访问节点的位置。

现给出 CirList 类的定义如下。

```
template <class T> class CirList{

    ListNode<T>* head;
    ListNode<T>* tail;
    ListNode<T>* cur;

public :
    CirList();
    ~CirList();

    bool AddTail(T value);
    void RemoveThis();
    void RemoveAll();
    void SetBegin();
    int GetCount();
    ListNode<T>* GetCur();

    bool IsEmpty();
    T GetNext();
};
```

其中，成员变量 head 表示表头指针，tail 表示表尾指针，cur 记录着当前被访问节点的位置。

函数 CirList()是默认的构造函数，函数~CirList()是析构函数。函数 AddTail(T value)用来向单向循环链表的表尾插入新节点。函数 RemoveThis()用来删除 cur 所指向的节点，

在删除完成后 cur 自动向后移动一个节点。函数 RemoveAll()用来删除链表中的所有节点，并在删除完成后将 cur 置于 head 处。函数 SetBegin()用来置 cur 到 head 处。在对单向循环链表进行访问前调用这个函数，可以确保访问从第 1 个节点开始。函数 GetCount()用来获得链表中节点的个数（表头节点除外）。函数 GetCur()用来返回指针 cur。函数 IsEmpty()用来判断链表是否为空，为空时返回值 1，非空时返回值 0。链表为空是指链表中除了表头节点外再没有其他节点。函数 GetNext()返回 cur 指向的节点中存储的数据，并在返回数据后将指针 cur 自动后移一个节点。如果对表尾应用 GetNext()，则返回表尾的值后 cur 回到链表中的第 1 个节点处，注意，不是表头节点处。

下面给出单向循环链表类中主要成员函数的实现代码，完整的源码清单可以从本书的在线支持资源中下载得到。

```
template <class T>
CirList<T>::CirList(){//默认的构造函数
     head = tail = new ListNode<T>; //新建节点，并初始化 head 与 tail
     cur = head;
     head->SetLink(head);
}

template <class T>
CirList<T>::~CirList(){
     RemoveAll();
     delete head;
}

template <class T>
bool CirList< T >::AddTail(T value){ //在链表尾插入新节点
    //新建节点，存储数据 value
     ListNode<T>* add = new ListNode<T>(value);

     tail->SetLink(add);        //把新节点链入链表
     tail = tail->GetLink();    //移动 tail 至新表尾处
     tail->SetLink(head);       //使表尾指针指向表头

     if(tail != NULL)
          return true;
     else
          return false;
}

template <class T>
void CirList<T>::RemoveThis(){//把 cur 指向的节点删除
     if(cur == head){
     //如果当前 cur 在 head 处，由于表头节点不可删除，顺序移动 cur 至下一节点处
        cur = cur->GetLink();
     }

     //用指针 preCur 标记 cur 的前一节点，node_del 标记要删除的节点
     ListNode<T>* preCur = cur;
     ListNode<T>* node_del = cur;

    //寻找 cur 的前一节点，并用 preCur 记录下来
```

```
    for(int i=0;i<this->GetCount();i++){
       preCur = preCur->GetLink();
    }

   //使 cur 前一节点的指针指向 cur 后一节点，即跳过 cur 所在的节点
    preCur->SetLink(cur->GetLink());
    delete node_del;
   //删除后 cur 顺序移动一个节点。
    cur = preCur->GetLink();
    preCur = NULL;
}

template <class T>
void CirList<T>::RemoveAll(){//删除所有节点
    SetBegin();//开始从第 1 个节点删除
    int length = GetCount();//获取链表长度
    for(int i=0;i<length;i++){
        RemoveThis();
    }
    cur = head;  //置 cur 到 head 处
}

template <class T>
void CirList<T>::SetBegin(){//置 cur 到 head 处，表示链表的开始
    cur = head;
}

template <class T>
int CirList<T>::GetCount(){//获得链表长度
    int num = 0;
    if (cur==NULL)
        this->SetBegin();
    ListNode<T>* here = cur;        //记录当前节点位置
    while(cur->GetLink() != here){ //遍历整个链表，记录节点个数
        cur = cur->GetLink();
        ++num;
    }
    cur = cur->GetLink();//使 cur 回到遍历前的位置
    return num;
}

//返回当前节点中的数据，并使 cur 顺序移动一个位置
template <class T >
T CirList< T >::GetNext(){
    if(cur == head){
      //跳过表头节点，因为表头节点中不存储有效数据
        cur = cur->GetLink();
    }
    T num = cur->GetData();//获得数据
    cur = cur->GetLink();   //顺序移动 cur 至下一个节点
    return   num;
}
```

利用以上介绍的单向循环链表类可以解决现实中的许多问题，其中一个经典的例子就是约瑟夫环问题。4.3.3 节将介绍单向循环链表在解决该问题上的应用。

4.3.3　约瑟夫环的问题

经典问题求解

约瑟夫环问题，读者也可以尝试用非循环链表来解决它。

弗拉维奥·约瑟夫（Flavius Josephus）是公元 1 世纪的一位著名的历史学家。他曾在自己的作品中记述了现在被普遍称为约瑟夫环问题的最初版本。所谓约瑟夫环问题，可简单地表述如下。

有 15 个人排成一圈，并给他们 1~15 的编号。现在从 1 号开始报数，报数字 4 的人退出队列，余下的人从退出者下一个位置开始继续刚才的报数，直到整个队列中只剩下一个人为止。请问这个人是几号？

分析：这个问题乍看起来好像很困难。由于每次都有人从队列中退出，所以队列的长度在不断变化，这使得整个报数的过程难以预测。但由问题的表述来看，约瑟夫环问题完全可以用单向循环链表来模拟。

下面给出示例程序的代码清单。

```
#include "stdafx.h"
#include "CirList.h"
#include <iostream>

using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{

    CirList<int> jos; //新建单向循环链表，模拟约瑟夫问题

    //向链表中加入 1~15 ，代表编号为 1~15 的人
    for(int i=1; i<16; i++){
        jos.AddTail(i);
    }
    jos.SetBegin();//开始模拟约瑟夫问题

    //记录原始队列的人数，用此人数减 1 即可得到要删去多少人
    //本题中要删去 14 人
    int length = jos.GetCount();
    for(int i=1; i<length; i++)
    {
        for(int j=0; j<3; j++)
            jos.GetNext();
        jos.RemoveThis();
    }

    cout<<jos.GetNext()<<endl;
    system("PAUSE");
    return 0;
}
```

完成编码后，编译并运行程序，输出结果为 13，即最后 13 号会被留下。从实现程序来看，这个算法的时间复杂度为$O(mn)$，其中n为人数。每回报m这个数的人退出队列。这意味着如果m和n都很大时，用此算法计算约瑟夫环问题的效率是十分低下的。恰当地运用一些数论知识可以有效地提高该问题的求解效率。下面给出另一种求解方案的核心算法。

```
for (int i=2; i<=n; i++){
    s=(s+m)%i;
}
```

其中，s 即为最后留在队列中的人。由于本章讨论的重点是链表问题，所以就不对上述方法做过多的讲述了。有兴趣的读者可以亲自测试一下，当m和n都很大时，以上这两种程序在执行效率上的差异。

经典问题求解

魔术师发牌问题，接下来就是见证奇迹的时刻！

4.3.4 魔术师发牌问题

魔术师利用一副牌中的 13 张黑桃，预先将它们排好后叠放在一起，牌面朝下。对观众说：我不看牌，只数数就可以猜到每张牌是什么，我大声数数，你们边听边看，接下来就是见证奇迹的时刻。魔术师将最上面的那张牌数为 1，把它翻过来正好是黑桃A，将黑桃A置于桌子上。然后按顺序从上到下数手上的余牌，第 2 次数 1、2，将第 1 张牌放在这些余牌的下面，而将第 2 张牌翻过来，正好是黑桃 2，将它放在桌子上。第 3 次数 1、2、3，将前面两张依次放在余牌的下面，再翻第 3 张牌正好是黑桃 3。这样依次进行将 13 张牌全翻出来，准确无误。请问魔术师手中的牌原始顺序是怎样的？

本问题的求解策略比较简单。先建立长度为 13 的链表并将链表中的每个元素置为 0，表示链表中什么牌也不存放。接下来根据魔术师发牌的顺序把 13 张牌存入链表中。例如，第 1 张牌存入第 1 个位置，第 2 张牌存入距第 1 张牌两个空位的位置。注意，在存牌前只需记录当前链表中空位的个数并根据这个数字存放新牌。若一个位置已经被某张牌占据，则这个位置不可以再存放新牌。

下面给出示例程序的代码清单。

```
#include "stdafx.h"
#include "CirList.h"
#include <iostream>

using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
    CirList<int> poker;
    for(int i=0;i<13;i++){
        poker.AddTail(0);     //创建循环链表，存储 13 张扑克牌
    }

    poker.SetBegin();
    poker.GetCur()->GetLink()->SetData(1);
    for(int i=2;i<14;i++){
        for(int j=0;j<i;j++){
            poker.GetNext();  //寻找插牌位置
            if(poker.GetCur()->GetData() != 0){
                j--;          //若当前位置中已有牌，则顺序查找下一个位置
            }
        }
        poker.GetCur()->SetData(i);
    }

    poker.SetBegin();
```

```
    for(int i=0;i<13;i++){
        cout<<poker.GetNext()<<"*";
    }
    cout<<endl;

    system("PAUSE");
    return 0;
}
```

完成编码后，编译并运行程序即可。

4.3.5　拉丁方阵的问题

经典问题求解

拉丁方阵问题，拉丁方阵在统计实验设计中具有重要应用。

拉丁方阵，或称为拉丁方，是一种特殊的N阶方阵。如果用从 1 开始的N个连续正整数排成$N \times N$的方阵，且每一行和每一列都没有重复的数，就称其为一个N阶拉丁方阵。因为这样的方阵最早填充的是拉丁字母，因此得名拉丁方阵。例如，下面给出的是一个 4 阶拉丁方阵。

$$\begin{matrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 1 \\ 3 & 4 & 1 & 2 \\ 4 & 1 & 2 & 3 \end{matrix}$$

本问题可以利用循环链表来求解。首先，初始化链表长度为矩阵阶数并在链表中存放从1~N连续的自然数。然后从链表头遍历链表并输出链表节点中的数字，以便构成第 1 行；第 2 行的构成方式与第 1 行类似，但此时遍历链表时并不从链表的第 1 个节点开始遍历，而是从链表的第 2 个节点开始；第 3 行的构成方式与第 2 行类似，但此时从链表的第 3 个节点开始遍历链表。重复利用以上方法最终构成整个拉丁方。

下面给出示例程序的代码清单。

```
#include "stdafx.h"
#include "CirList.h"
#include <iostream>

using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
    int num;
    cout<<"请输入拉丁方阶数(2<=N<=9):";
    cin>>num;
    CirList<int> latin;
    for(int i=1;i <= num; i++){
        latin.AddTail(i);                    //创建循环链表
    }

    latin.SetBegin();
    for(int i=1; i<=num; i++){
        for(int j=1; j<=num; j++){
            cout<<latin.GetNext()<<" ";     //输出循环链表中的数据
        }
        cout<<endl;
        latin.GetNext();                     //顺序移动循环链表中的所有数据
    }
```

```
    system("PAUSE");
    return 0;
}
```

完成编码后，编译并运行程序即可。

4.4 双向循环链表

利用前面介绍的单向链表与单向循环链表可以方便地找出已知节点的后继节点：在单向链表中，只要指针 link 非空，就可以根据指针 link 找到已知节点的后继；在单向循环链表中同样可以这么做，但需注意避免死循环的发生。在以上两种链表中寻找已知节点的前驱节点是十分不容易的：必须从链表头开始沿 link 顺序遍历链表，直到某一节点的后继节点为指定节点为止。当链表中节点的数目很多时，用这种方法寻找前驱节点的效率并不理想。双向循环链表就是为了克服这一缺点而被引入的。

4.4.1 双向循环链表的结构

在双向循环链表中，每个节点中都储存了两个指针，一个指示该节点的前驱节点，另一个指示该节点的后驱节点。一个双向循环链表有一个表头节点，它同样包含有两个指针，一个指向链表的最后一个节点，一个指向链表最前端的第 1 个节点。链表的最后一个节点同样包含有两个链接指针，一个指向链表的表头节点，另一个指向链表的倒数第 2 个节点。双向循环链表的结构如图 4-18 所示，双向循环链表正是通过这种双指针形式实现链表的双向循环的。为符合双向循环链表的这种特殊结构，在类的实现上，双向循环链表类中的节点除了要封装指向后继节点的指针外，还要封装一个指向前驱节点的指针 prior。

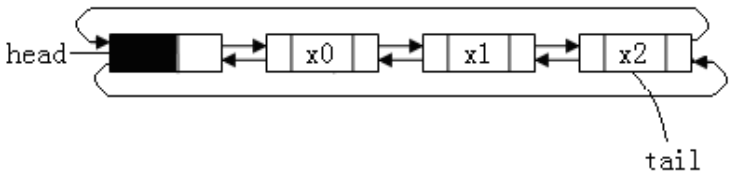


图 4-18 双向循环链表结构

双向循环链表的特点是可以沿两个方向对整个链表进行遍历，在每个方向上，双向循环链表都表现出单向循环链表的性质。由于节点封装了指向前驱的指针 prior，所以寻找已知节点的前驱节点是十分方便的，只需沿着 prior 指针指向的方向寻找即可，算法的时间复杂度为$O(1)$。

连接两个不同的双向循环链表是比较复杂的，具体过程如图 4-19 所示。首先将第 1 个链表的表头节点中指向该链表最后一个节点的指针改为指向第 2 个链表的最后一个节点。再将第 2 个链表的最后一个节点中指向该链表表头节点的指针改为指向第 1 个链表的表头节点。然后将第 1 个链表的最后一个节点中指向该链表表头节点的指针改为

指向第 2 个链表的第 1 个节点，并将第 2 个链表的第 1 个节点中指向该链表表头节点的指针改为指向第 1 个链表的最后一个节点。最后删除第 2 个链表的表头节点。如此，整个连接过程就完成了。

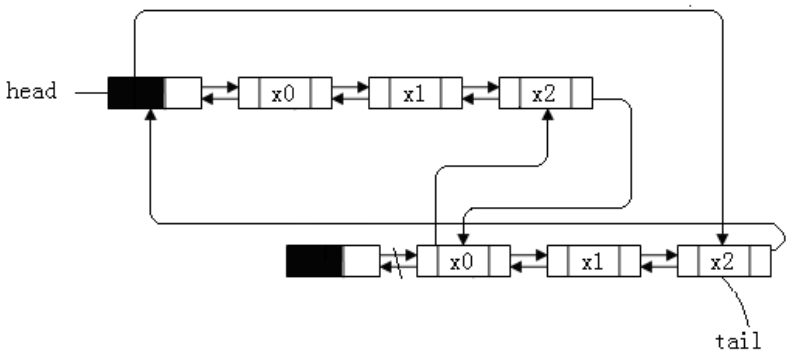


图 4-19　双向循环链表的连接

由前文可知，在单向循环链表中可用表尾指针 tail 代替表头指针 head。而在双向循环链表中可用表尾指针 tail 代替表头指针 head, 也可用表头指针 head 代替表尾指针 tail。这是因为链表中的循环是双向的，表头指针 head 与表尾指针 tail 只要给定其中一个，总可以找到另一个，算法的时间复杂度均为$O(1)$。

与其他链表一样，双向循环链表在内存中的存储也不必是连续的。各个节点可以被分散地存储在内存中。

4.4.2　双向循环链表的实现

在双向循环链表中，节点类既要封装指向后继节点的指针 link，还要封装指向前驱节点的指针 prior, 所以类中要增加一个新的成员变量 prior。同时还要添加方法 GetPrior()和 SetPrior(ListNode<T>*next)，这两个方法的作用与 GetLink()和 SetLink (ListNode <T>* next)正好相反：GetPrior()返回指向前驱节点的指针，SetPrior(ListNode<T>* next)改变指针 prior 的指向。

现给出 DouListNode 类的定义如下。

```
template <class T > class DouListNode{

    T data;
    DouListNode<T>* link;
    DouListNode<T>* prior;

public :

    DouListNode() : link(NULL),prior(NULL){}
    DouListNode(T value) : link(NULL),prior(NULL),data(value){}
    ~DouListNode(){};
    void SetLink(DouListNode<T>* next);
    void SetPrior(DouListNode<T>* pre);
    DouListNode<T>* GetLink();
    DouListNode<T>* GetPrior();
```

```
    T& GetData();
};
```

请注意 DouListNode 类的构造函数 DouListNode()及 DouListNode(T value)。在这两个构造函数的初始化列表中不但要初始化指针 link 为“ NULL ”，还要对指针 prior 进行相同的初始化。

函数 SetLink(DouListNode<T> * next) 、GetLink ()及 GetData()的作用均与前面相同，这里不再赘述。限于篇幅，此处不再详细列出类 DouListNode 中函数的实现代码，完整代码清单可从本书在线支持资源中下载得到。

在介绍完上述节点类 DouListNode 后，下面介绍双向循环链表类 DouList。与前文所述的两种链表类似，当一个双向循环链表为空链表时，它至少保留一个表头节点。此时表头节点中保存的指针 link 与 prior 均指向这个表头节点自身，如图 4-20 所示。

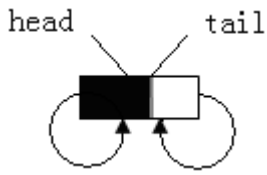


图 4-20　空双向循环链表

在双向循环链表的结尾插入一个节点需要同时改动 5 个指针：新建节点中指针类数据成员 link 与 prior，原链表表尾节点中的指针 prior，表头节点中的指针 link 及表尾指针 tail。首先将目标链表的最后一个节点中指向表头节点的指针改为指向待插入节点，然后将待插入节点的 prior 指针指向目标链表中的最后一个节点。再将待插入节点的 link 指针指向目标链表中的表头节点，并将目标链表的表头节点中指向最后一个节点的指针改为指向待插入节点。具体插入过程见图 4-21。

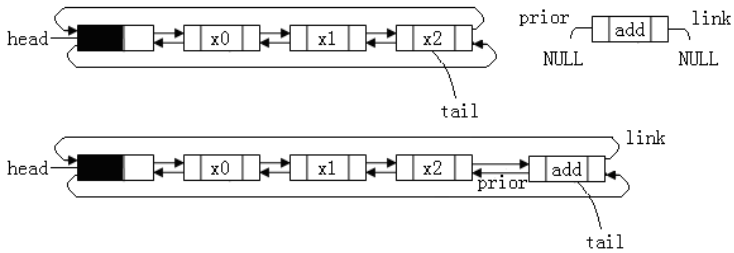


图 4-21　在表尾插入新节点

若不在表尾插入，则情况比较简单，只需要改动 4 个指针即可，具体过程见图 4-22。

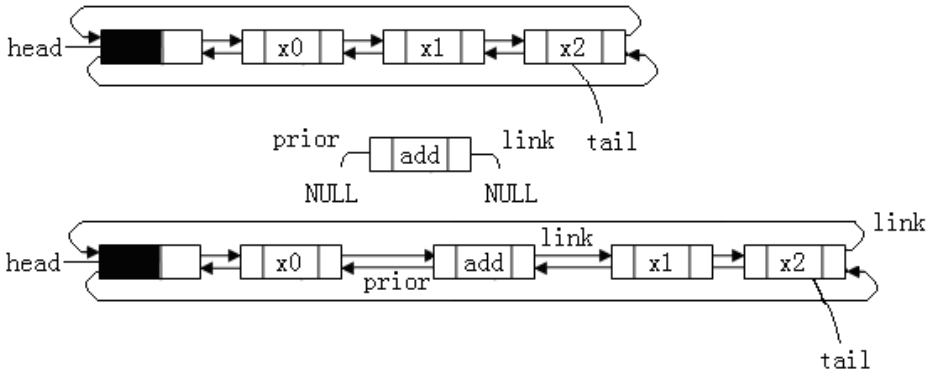


图 4-22　在其他位置插入节点

在删除操作中，若被删节点位于表尾，则删除过程如图 4-23 所示。该过程首先将目标链表的表头节点中指向表尾节点的指针改为指向链表的倒数第 2 个节点，然后将倒数第 2 个节点中指向表尾节点的指针改为指向链表的表头节点，再将待删除节点释放即可。

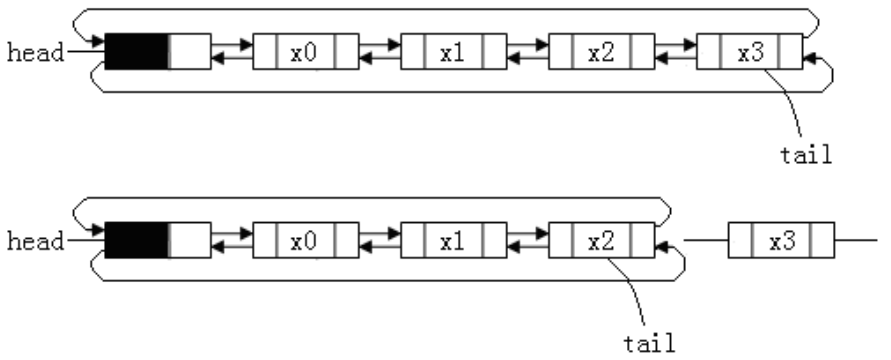


图 4-23　删除表尾节点

如果这个节点不是表尾，则删除过程如图 4-24 所示。首先将待删除节点的前驱节点中指向待删除节点的指针改为指向待删除节点的后继节点，然后将待删除节点的后继节点中指向待删除节点的指针改为指向待删除节点的前驱节点，再将待删除节点释放即可。

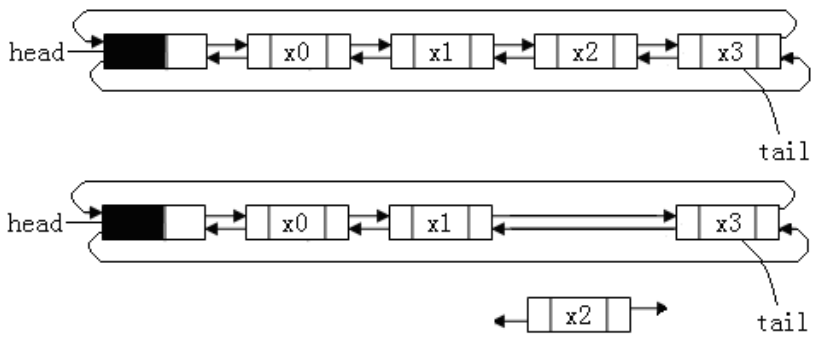


图 4-24　在其他位置删除节点

在双向循环链表中，由于可以从两个不同的方向对链表进行遍历，所以双向循环链表类 DouList 要增加成员函数 GetPrior()。它的作用是返回当前指针 cur 指向的节点中的数据，并在成功返回数据后使 cur 指向其前驱节点。

对应于 AddTail(T value)，在类 DouList 中添加函数 AddHead(T value)。它用来向表头节点与链表中第 1 个节点之间插入新节点。

函数 InsertAfter(T value)的作用是向 cur 指向的节点后插入一个数据成员为 value 的新节点。当 cur == tail 时，这个函数的作用相当于 AddTail(T value)。

函数 void TowardCur()与 void BackCur()的作用分别是使指针 cur 向后继节点方向与前驱节点方向移动。与 GetNext()和 T GetPrior()不同的是，这两个函数不返回节点中的数据。

函数 RemoveThis()需要重载。重载后的函数应该能接收一个参数 bool direction，这个参数指向了当 cur 所指向的节点被删除后 cur 移动的方向。在本章中，当 direction == 1 时，cur 顺着 link 指针方向移动；当 direction == 0 时，cur 顺着 prior 指针方向移动。

双向循环链表的判空条件与单向循环链表的判空条件相同，即只要判断表头节点的后继节点是否仍为表头节点。还可以反过来判断表头节点的前驱节点是否仍为表头节点，如果答案是肯定的，则链表为空。注意这两个条件其实是等价的，所以只要判断一个就可以了，无须两个条件都判断。

现给出 DouList 类的定义如下。

```
template <class T > class DouList{
    DouListNode<T>* head;
    DouListNode<T>* tail;
    DouListNode<T>* cur;

public :
    DouList();
    ~DouList();

    bool AddTail(T value);
    bool AddHead(T value);
    void RemoveThis(bool direction);
    void RemoveAll();
    void SetBegin();
    int GetCount();
    void TowardCur();
    void BackCur();
    DouListNode<T>* GetCur();
    DouListNode<T>* GetHead();
    DouListNode<T>* GetTail();
    void InsertAfter(T value);

    bool IsEmpty();
    T GetNext();
    T GetPrior();
};
```

函数 AddTail()用来在链表尾部插入一个新节点。函数 RemoveAll()用来删除链表中的

所有节点（但保留表头节点）。函数 SetBegin()用来将 cur 置于表头处。函数 GetCount()用来返回链表中的节点数(表头节点除外)。函数 GetCur()用来返回指针 cur。函数 IsEmpty()用来判断链表是否为空(即只有表头节点)。函数 GetHead()与 GetTail()分别用来返回头指针与尾指针。完整的双向循环链表类 DouList 的实现代码清单可从本书的在线支持资源中下载得到，此处不具体列出。

4.4.3　维吉尼亚加密法问题

经典问题求解

维吉尼亚加密法问题，一种古老的暗语传递方式。

维吉尼亚（Vigenere）加密法问题是古典加密法中一种重要的方法，后来很多的加密法都借鉴了它的思想。这种加密法相对于其他的古典加密法来说是比较安全的，因此曾经被广泛使用。

究竟该加密法是如何工作的呢？在密码学中，未被加密的原始信息称为明文，加密后不可读的信息称为密文，而用来把明文转换成密文的工具叫作密钥。维吉尼亚加密法的工作原理很简单，先看一个例子：众所周知，英语中共有 26 个字母。现在取出字母 a，把这个a向字母表后面移动 3 个位置，于是a便成了d。如果对单词 here 进行相同的操作，则它将变成 khuh。可见原来可读的单词变成了不可读的信息，这就实现了对明文的加密。如果单词中含有x、y、z该怎么处理？把这些字母向字母表后面移动 3 个位置则超出了字母表的范围。注意，这里的字母表是一个“循环”的字母表，把z向后移 3 个字母会成为c。类似地，y成为b，x成为a。当然，上述步骤是维吉尼亚加密法处理单个字符的过程。在处理具体密文的时候，维吉尼亚加密法首先需要一个密钥，该密钥以一个向量的形式存在。假设该向量的长度为 6，那么接着就在一定的数值范围内选择相应数目的整数来填满这个密钥向量。例如，我们可以从 1~10 范围内随机选择 6 个整数作为密钥，即 6、9、8、5、9、2。密钥向量中的元素是可以重复的，但过度的重复会降低密码的安全性。然后取出明文的第 1 个字母并将之移动 6 位，第 2 个字母移 9 位，一旦到了密钥的末尾，又从头开始。也就是说，第 7 个字母还是移 6 位，第 8 个字母移 9 位。如此循环下去直至把明文加密完毕。如下所示为一个维吉尼亚加密法的例子，其中第 1 行是明文，第 2 行是循环使用的密钥，第 3 行是加密处理后得到的密文。

y	o	u	a	r	e	i	n	d	a	n	g	e	r
6	9	8	5	9	2	6	9	8	5	9	2	6	9
e	x	c	f	a	g	o	w	l	f	w	i	k	a

下面的程序实现了运用维吉尼亚加密法对明文进行加密处理的方法。注意在加密时我们将忽略空格、数字或者标点符号，对于字母我们也默认仅处理小写字母的情况。这个例子很好地演示了 4.4.2 节中我们创建的双向循环链表类的使用。

```
#include "stdafx.h"
#include "DouListNode.h"
#include "DouList.h"
#include <iostream>

using namespace std;
```

```
int _tmain(int argc, _TCHAR* argv[])
{
    DouList<char> planText;
    DouList<char> cryptograph;
    DouList<int> key;
    DouList<char> trans;
    planText.SetBegin();
    //将明文字符逐个填入链表 planText, 例如 planText.AddTail('y')
    //详细内容此处略去

    planText.SetBegin();
    cout<<"明文:"<<'\t';
    for(int z=0;z<planText.GetCount();z++){
        cout<<planText.GetNext()<<" ";
    }
    cout<<endl<<endl;

    key.SetBegin(); //产生密钥链表
    for(int i=0;i<6;i++){
      key.AddTail(1+rand()%9);
    }

    cout<<"密钥:"<<'\t';
    for(int i=0;i<key.GetCount();i++){
        cout<<key.GetNext()<<" ";
    }
    cout<<endl<<endl;

    planText.SetBegin();
    key.SetBegin();
    cryptograph.SetBegin();
    for(int i=0;i<planText.GetCount();i++){

        char c = planText.GetNext();
        int num = key.GetNext();
        if('a'<=c&&c<='z'-num)
            c=c+num;
        else if('z'-num<c&&c<='z')
            c = c+num-1-'z'+'a';
        cryptograph.AddTail(c);
    }

    cryptograph.SetBegin();
    cout<<"密文:"<<'\t';
    for(int j=0; j<cryptograph.GetCount(); j++){
        cout<<cryptograph.GetNext()<< " ";
    }
    cout<<endl<<endl;

    trans.SetBegin();
    planText.SetBegin();
    key.SetBegin();
    for(int k=0; k<planText.GetCount(); k++){


        char c = cryptograph.GetNext();
        int num = key.GetNext();
        if('a'<=c-num && c-num<='z')
            c=c-num;
```

```
        else if('a'>c-num && c>='a')
            c = 'z'-('a'-c+num)+1;

        cryptograph.AddTail(c);
        trans.AddHead(c);
    }

    trans.SetBegin();
    cout<<"解密:"<<'\t';
    for(int k=0;k<trans.GetCount();k++){
        cout<<trans.GetPrior()<<" ";
    }
    cout<<endl<<endl;

    system("PAUSE");
    return 0;
}
```

上面的程序中，默认的密钥空间 1~10，密钥长度为 6。在实际应用中可以加大密钥空间，改变加密顺序，或是进行多次加密等。这些措施都可以使得密文更加难以破译，从而提高传输安全性。维吉尼亚加密法的安全性在于密钥不可泄露，一旦密钥泄露攻击者就可以对密文进行破译。

4.5　游标类的设计与实现

为了方便对链表进行访问，不妨引入一个新的类——游标类，这种处理方式在 STL 的实现中非常普遍。

4.5.1　游标类的结构

当要遍历一个链表时（本节仅用单向循环链表来举例），可以考虑下面的实现方法。

```
for(int i=0; i<MyList.GetCount(); i++){
    cout<< MyList.GetNext()<<"*";
}
```

用这种方法虽然可以对链表进行遍历，但代码可读性并不强。类 List 中虽然封装了很多方法，但并没有哪一个方法是专门用来完成遍历功能的。之所以不提供这些方法是因为类 List 要尽量保持精简，如果把这些方法都封装到类 List 中去，势必让实现代码变得臃肿。面对未来出现的多种多样的新要求，必须不断地修改类的定义，这样就降低了代码的可复用性。

解决以上难题的一种简单有效的方法就是把对链表执行遍历功能的函数专门封装到另外的一个类中。这个类就是游标类（Iterator）。游标类（Iterator）中封装的方法不但能更容易地遍历链表，而且使用这些方法的代码将变得简洁易懂。当链表中的内容改变时，如果有必要，只需要修改游标类（Iterator）中的方法而不必修改链表类，这样就有利于建立链表类的类库，有利于代码复用。

下面给出游标类（Iterator）的定义如下。

```
template <class T> class Iterator{

    ListNode<T>* cur;

    public:
        Iterator():cur(NULL){};
        Iterator(ListNode<T>* point):cur(point){}

        friend T operator ++(Iterator& it);         //前缀
        friend T operator ++ (Iterator& it,int);    //后缀

        friend bool operator == (Iterator& it,ListNode<T>* point);
        friend bool operator !=  (Iterator& it,ListNode<T>* point);
        friend ostream& operator<< (ostream& stream,Iterator& it);

        void operator= (ListNode<T>* point);
        ListNode<T>*& GetCur();
};
```

4.5.2 游标类的实现

从 4.5.1 节中给出的游标类的定义可得出它之所以能提高代码的可读性是由于使用了运算符重载。重载的运算符使得对游标类的操作异常简单。例如，可以用下面的代码实现对链表的遍历。

```
Iterator<int> a;
a = listFirst.GetHead();
while(a!=listFirst.GetTail()){
    ++a;
    cout<<a;
}
```

这段代码首先建立 int 型游标a，然后把语句 listFirst.GetHead()的返回值赋给a，这样游标a便指向链表的表头节点。语句++a使游标a向链表的尾部移动。语句 cout<<a输出游标a指向的节点中的数据。至于函数的具体实现过程将在后文中具体介绍。

游标类（Iterator）中有一个私有的数据成员 cur，这是一个指向 ListNode 类的指针，可见游标类也是利用指针完成对链表的遍历的。

带一个参数的构造函数 Iterator(ListNode<T>* point)初始化 cur 为 point。而默认的构造函数 Iterator()初始化 cur 为 NULL。

重载运算符 operator ++(Iterator& it)作为前缀使用，它首先使 cur 向后移动一个节点，然后返回 cur 所指向节点中的数据。重载运算符 operator ++ (Iterator& it,int)作为后缀使用，它先返回 cur 所指节点中的数据，再使 cur 向后移动一个节点。

重载运算符 operator == (Iterator& it,ListNode<T>* point)用来比较 cur 与参数 point 是否相等, 若相等则返回 1, 不相等返回 0。重载运算符 operator != (Iterator& it,ListNode<T>* point)的作用与前者运算符相反：若 cur 与 point 相等则返回 0，不相等则返回 1。

重载的输出运算符 ostream& operator<< (ostream& stream,Iterator& it)用来输出 cur 指向节点中的数据。重载的赋值运算符 operator= (ListNode<T>* point)用来把参数 point 的

值赋给 cur。函数 GetCur()用来返回 cur 指针。

以下是游标类 Iterator 中各方法的实现部分。

```
//前缀运算符，先返回 cur 所指向节点中的数据
//再使 cur 向后移动
template <class T>
T operator++(Iterator<T>& it){
it.GetCur() = it.GetCur()->GetLink();
T data;
data = it.GetCur()->GetData();
     return data;
}

//后缀运算符，先使 cur 向后移动一个节点
//再返回 cur 所指向节点中的数据
template <class T>
T operator++(Iterator<T>& it,int){
     T data;
     data = it.GetCur()->GetData();
     it.GetCur() = it.GetCur()->GetLink();
     return data;
}

//判断 cur 是否等于 point
template <class T>
bool operator==(Iterator<T>& it,ListNode<T>* point){
     return it.GetCur() == point;
}

//判断 cur 是否不等于 point
template <class T>
bool operator!=(Iterator<T>& it,ListNode<T>* point){
     return it.GetCur() != point;
}

//输出 cur 所指向节点中的数据
template <class T>
ostream& operator<<(ostream& stream,Iterator<T>& it){
     stream<<it.GetCur()->GetData();
     return stream;
}

//把 point 的值赋给 cur
template <class T>
void Iterator<T>::operator = (ListNode<T>* point){
     cur = point;
}

//返回 cur
template <class T>
ListNode<T>*& Iterator<T>::GetCur(){
     return cur;
}
```

下面就运算符 operator++(Iterator<T>& it)与 operator++(Iterator<T>& it,int)的区别进行讲解。前者重载的运算符做前缀使用。例如，语句“ ++a ”，其作用是先把 cur 向后移

动一个节点的位置，然后再返回 cur 指向的节点中的数据。换句话说，函数的实现部分将首先执行语 it.GetCur()=it.GetCur()->GetLink()，然后再执行语句 data=it.GetCur()->GetData()。

重载运算符 operator++(Iterator<T>& it,int)作为后缀使用，这导致语句执行时要先返回 cur 所指节点中的数据，所以函数的实现部分首先执行语句 data=it.GetCur()->GetData()，再执行语句 it.GetCur()=it.GetCur()->GetLink()。请注意该重载运算符表达式中的第 2 个参数，这个 int 型的参数并没有被函数体使用，编译器为这个参数传递一个哑元常量值，作用是区别++运算符是作为前缀使用还是作为后缀使用。下面再来回顾前面提到的代码。

```
Iterator<int> a;
a = listFirst.GetHead();
while(a!=listFirst.GetTail()){
    ++a;
    cout<<a;
}
```

这段代码首先用赋值运算符=使游标类中的指针 cur 指向链表的首节点，然后进入循环语句 while。进入循环后语句++a 使 cur 不断后移，同时输出 cur 所指向节点中的数据，从而实现对链表的遍历。利用游标类对链表进行遍历的方式不止一种，例如，下面的代码同样可以实现对链表的遍历。

```
Iterator<int> a;
a = listFirst.GetHead()->GetLink();
while(a!=listFirst.GetTail()->GetLink()){
    cout<<a++;
}
```

这段代码首先使游标a指向链表中的第 1 个节点，然后进入 while 循环。进入循环后首先输出游标a所指向的节点中的数据，然后再把a向后面的节点移动。

以上介绍的游标类仅适用于遍历单向链表类，如果要遍历单向循环链表与双向循环链表类应该编写各自的游标类，这些游标类的定义与实现均与以上游标类相类似。下面仅给出双向循环链表类的定义与实现。

双向链表游标类定义如下。

```
template <class T> class Iterator{

    DouListNode<T>* cur;

    public:
        Iterator():cur(NULL){};
        Iterator(DouListNode<T>* point):cur(point){}
        friend T operator++(Iterator& it);          //前缀
        friend T operator++(Iterator& it,int);      //后缀
        friend T operator--(Iterator& it);          //前缀
        friend T operator--(Iterator& it,int);      //后缀
        friend bool operator==(Iterator& it,DouListNode<T>* point);
        friend bool operator!=(Iterator& it,DouListNode<T>* point);
        friend ostream& operator<< (ostream& stream,Iterator& it);
```

```
        void operator= (DouListNode<T>* point);
        DouListNode<T>*& GetCur();
};
```

双向链表游标类的实现部分如下。

```
template <class T>
T operator++(Iterator<T>& it){
    it.GetCur() = it.GetCur()->GetLink();
    T data;
    data = it.GetCur()->GetData();
    return data;
}

template <class T>
T operator++(Iterator<T>& it,int){
    T data;
    data = it.GetCur()->GetData();
    it.GetCur() = it.GetCur()->GetLink();
    return data;
}

template <class T>
T operator--(Iterator<T>& it){
    it.GetCur() = it.GetCur()->GetPrior();
    T data;
    data = it.GetCur()->GetData();
    return data;
}

template <class T>
T operator--(Iterator<T>& it,int){
    T data;
    data = it.GetCur()->GetData();
    it.GetCur() = it.GetCur()->GetPrior();
    return data;
}

template <class T>
bool operator==(Iterator<T>& it,DouListNode<T>* point){
    return it.GetCur() == point;
}

template <class T>
bool operator!=(Iterator<T>& it,DouListNode<T>* point){
    return it.GetCur() != point;
}

template <class T>
ostream& operator<<(ostream& stream,Iterator<T>& it){
    stream<<it.GetCur()->GetData();
    return stream;
}

template <class T>
void Iterator<T>::operator = (DouListNode<T>* point){
    cur = point;
}

template <class T>
DouListNode<T>*& Iterator<T>::GetCur(){
```

```
    return cur;
}
```

由代码可见，双向循环链表游标类的实现部分几乎与单向链表相同，但由于双向循环链表可以在两个方向上遍历，它的游标类中增加了运算符“ -- ”，它的使用方法如下。

```
Iterator<int> a=list.GetTail();
while(a!=list.GetHead()){
    cout<<a--;
}
```

自减运算符“ -- ”使得指针 cur 向链表头方向移动。运算符“ -- ”也有前缀用法与后缀用法，这些用法均与“ ++ ”运算符类似，这里不再赘述。

4.6 STL 与链表

C++标准模板库中提供了一个链表类 list，它是标准模板库中众多容器的一种。这个链表类提供了丰富的函数，用户利用这些函数可以方便地对链表进行操作。

4.6.1 STL 中链表类的接口

链表类 list 支持对节点进行插入与删除操作。这些操作既可以在表尾进行，也可以在表头进行。具体来说，函数 push_front 用于向链表的第 1 个节点之前插入一个新节点；函数 pop_front 用于删除链表的第 1 个节点；函数 push_back 用于在表尾添加一个节点；函数 pop_back 用于删除表尾节点。除了添加或删除节点以外，链表类 list 也支持对节点中数据的访问，其中函数 front 用于访问链表中第 1 个节点的数据；函数 last 用于访问链表表尾节点的数据。

下面给出一段示例代码。

```
#include "stdafx.h"
#include <iostream>
#include <string>
#include <cstdlib>
#include <list>

using namespace std;

int _tmain(int argc, _TCHAR* argv[]) {
    list<int> myList1;
    myList1.push_front(10);
    cout << myList1.size() << endl; //链表长度为 1

    list<string> myList2 (10);
    cout << myList2.size() << endl; //链表长度为 10

    list<double> myList3 (2, 4.6);   //初始值为 4.6
    cout << myList3.back() << endl;
    myList3.pop_back();
    cout << myList3.empty() << endl;

    system("PAUSE");
```

```
    return 0;
}
```

在使用链表类（list）前，应在文件开始处包含如下代码。

```
#include <list>
```

在上面的代码中，函数 size()与 empty()分别用来计算链表长度与判断链表是否为空。这两个函数类似于前面类 List 中介绍的函数 GetCount()与 IsEmpty()。

除了默认的构造函数，STL 中的链表类（list）还提供了一个重载的构造函数。利用这个构造函数用户可以指定链表中节点数据的初始个数。例如：

```
list<string> myList2(10)
```

这条语句就使用了链表类（list）中经过重载的构造函数，它创建了一个包含 10 个节点的链表。需要说明的是，这个重载的构造函数还能接收第 2 个参数，开发人员可以利用这第 2 个参数指定链表节点中数据成员的初值。例如：

```
list<double> myList3(2, 4.6);
```

这条语句创建了一条用于存储 double 型数据的链表，这个链表包含有两个节点，且这两个节点的初始值都为 4.6。

为了使读者更方便地使用 STL 链表类 list, 下面把类 list 的主要参数说明和成员函数列举如下，见表 4-1、表 4-2。

表 4-1　参数说明表

参　　数	描　　述
value_type;	对象类型 T，存储在 list 中
pointer;	指向 T 的指针
reference;	T 的引用
const_reference;	T 的常量引用
size_type;	正整数类型
difference_type;	整数类型
iterator ;	访问 list 的迭代器
const_iterator;	访问 list 的常量迭代器
const_reverse_iterator;	访问 list 的反向迭代器

表 4-2　主要成员函数列表

函　　数	说　　明
iterator begin();	返回 list 的头指针
iterator end();	返回 list 的尾指针
reverse_iterator rbegin();	返回反向 list 的反向头指针
reverse_iterator rend();	返回反向 list 的反向尾指针
size_type size();	返回 list 的元素数量
bool empty();	判断 list 是否为空
list<Value>% operator=(list<Value>% right);	=运算符重载

续表

函　　数	说　　明
bool operator==(list<Value>% left, list<Value>% right);	重载==运算符
bool operator<(list<Value>% left, list<Value>% right);	小于运算符
bool operator>(list<Value>% left, list<Value>% right);	大于运算符
reference front();	返回第一个元素
reference back();	返回最后一个元素
void push_back(value_type val);	在 list 尾部插入一个元素
void pop_back();	删除最后一个元素
void swap(list<Value>% right);	交换两个 list 的内容
iterator insert(iterator pos, value_type val);	在 pos 前插入 val
void insert(iterator pos, size_type n, value_type val);	在位置 pos 前插入 *n* 个 val
iterator erase(iterator pos);	删除在位置 pos 的元素
iterator erase(iterator first, iterator last);	删除在[first, last]之间的元素
void clear();	删除所有的元素
void resize(size_type n, value_type val);	插入或删除使元素个数为 *n*, 插入的值为 val

4.6.2 遍历

对链表进行遍历有一个很有效的方法就是使用游标类。利用链表类（list）中提供的函数 begin()与 end()返回的游标，配合使用一个循环结构（例如 for 循环或 while 循环），就可以轻松地实现对链表的遍历。例如，下面这段代码片段。

```
//添加元素
list<int> myList;
for (int i = 0; i < 5; i++){
    myList.push_back(i);
}

//遍历链表
list<int>::iterator tra = myList.begin();
while (tra != myList.end()) {
    cout << *tra << endl;
    ++tra;
}
```

可能有的读者已经注意到，当变量 tra 自增到指向表尾节点时已不再满足条件 tra != myList.end()，于是循环结束。如果这样表尾节点不是永远也无法被访问到了吗？其实语句 myList.end()返回的并不是表尾节点的游标，而是表尾节点“后继”节点的游标。但语句 myList.begin()返回的却是链表中第 1 个节点的游标。这样链表中所有的节点就都可以被访问到了。

STL 链表类（list）是一个双向循环链表类，这意味着链表类（list）支持从表尾到表头的循环。下面这段代码片段演示了从表尾到表头进行遍历的方法。

```
//添加元素
```

```
list<int> back;
for (int i = 0; i < 10; i++) {
   back.push_back(i);
}

//逆向遍历
list<int>::iterator backTra = back.end();
do {
    -- backTra;
   cout << *backTra << endl;
}while (backTra != back .begin());
```

上面的代码虽然也能对链表进行遍历，但这种遍历方式显然没有正向遍历清晰明了。为了使逆向遍历同正向遍历的实现方式保持一致，游标 backTra 应该能够指向链表中的最后一个元素和第 1 个元素的前驱。实际上，函数 rend()与 rbegin()就是返回上述游标的函数。由此我们便可采用新的遍历方式，示例代码如下。

```
// 添加元素
list<int> back;
for (int i = 0; i < 10; i++) {
   back.push_back(i);
}

//逆向遍历
list<int>::reverse_iterator it = back.rbegin();  //逆向游标类
while (it != back.rend()) {
    cout << *it << endl;
    ++ it;
}
```

由函数 rbegin()与 rend()返回的游标属于逆向游标，这种游标适合于用来对链表进行逆向遍历。对逆向游标类的操作类似于普通游标类。但要注意的是，如果对逆向游标应用自增运算符（++），游标会逆着链表的方向移动一个节点。这与对普通游标类应用自增运算符（++）得到的效果完全相反。

除了链表类 list，STL 其他的容器（如队列 deque 等）亦提供逆向游标。逆向游标是对容器进行逆向遍历的有效工具。

4.6.3　元素的插入与删除

利用链表类 list 能有效地对节点进行插入与删除操作而无须过多关心节点位置，这是由链表连续性所决定的。在链表类 list 中成员函数 insert()与 erase()用来完成对节点的插入与删除。为了指明插入与删除的范围，这两个函数均利用了游标类。函数 erase()有两个版本：一个版本只接收一个参数，这个参数指明了要删除的元素；另一个版本接收两个参数，在这两个参数划定的范围内所有元素将被删除。下面给出一段应用函数 erase()的程序片段。

```
#include "stdafx.h"
#include <iostream>
#include <string>
#include <cstdlib>
#include <list>
```

```
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
    string str[] = {"Today ","is ","not ","Friday."};
    list<string> eraseThird;
    for(int i = 0; i < 4; i++)
        eraseThird.push_back(str[i]);

    //删除链表中的第 3 个元素
    list<string>::iterator third = eraseThird.begin();
    ++third;
    ++third;
    eraseThird.erase(third);

    list<string>::iterator it = eraseThird.begin() ;
    while (it != eraseThird.end()) {
        cout << *it ;
        ++it;
    }
    cout<<endl;

    system("PAUSE");
    return 0;
}
```

函数 insert()的用法与函数 erase()相似，下面给出应用函数 insert 的实例。

```
// 向第 3 个元素前插入新元素
list<string>::iterator third = insertThird.begin();
third ++ ;
third ++ ;
insertThird.insert(third, “not ”);
```

利用函数 insert()可以把一些连续的节点插入到已知链表中，具体的插入过程请见下面的程序片段。

```
// 向链表中插入连续节点
list<int> list1(15, 3);
list<int> list2(16, 2);
list<int>::iterator insertMass = list1.begin();
insertMass++;
list1.insert(insertMass, list2.begin(), list2.end());
cout << list1.size() << endl;
```

本节介绍了 STL 中链表类（list）的基本用法，本书后续的例子中还会用到该类。希望读者能够通过编程实践不断强化对它的掌握，熟练使用 STL 中的容器对每个 C++开发人员来说都是非常重要的。

本章参考文献

[1] 谭浩强．C 语言程序设计（第 2 版）．北京：清华大学出版社，2004.

[2] 殷人昆，陶永雷，谢若阳，盛绚华. 数据结构（面向对象方法与 C++描述）. 北京：清华大学出版社，1999.

第 5 章

先进先出与后进先出——简单而深刻

本章介绍栈与队列这两种数据结构。它们相互联系又互相区别，因此常常将它们放在一起讲解，并通过对比来体现各自的特点。把握二者区别的关键在于理解“ 先进先出 ”和“ 后进先出 ”的原则，本章将基于此展开讨论，并给出一些典型的经典问题，通过它们的求解将有助于加深读者对于这两种数据结构的理解。

5.1 摞盘子的策略

栈是一种非常重要的线性数据结构，它的特点在于最后入栈的数据项也将是最先被处理的数据项。本节将讨论栈的结构和它的两种实现方式，并在此基础上解决几个与栈有关的经典问题。

5.1.1 栈的结构

问题导入

从厨房里的盘子能否联想到栈结构的特点？

如果厨房里有一些散乱的盘子，你试图把它们整理在一起。于是你将其中一个摆放在某个平稳的位置上，假设标记这个盘子为P_0。然后选取另外一个P_1，并把它摞放在P_0上。再从散乱的盘子中选出P_2，并把它摞放在P_1上……如此继续下去直到原来散乱的盘子中最后一个盘子P_n被摞放到已经整理好的盘子上为止，原来散乱的盘子就变成了一叠摞好的盘子，如图 5-1 所示。

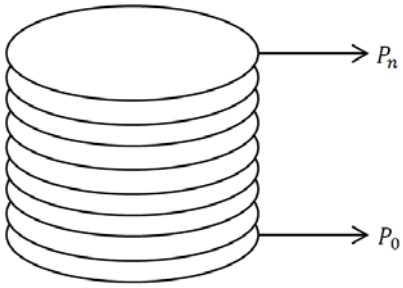


图 5-1 摞好的盘子

当你需要使用一个盘子的时候，想想你会怎么取盘子呢。显然从中间，或者从底部取盘子不是一个明智的决定，因为那样很容易将这叠盘子摔碎。最合理的想法就是总是从最上面开始取出盘子。这个例子是现实世界中一个模拟栈结构的典型例子。

栈是计算机科学中一种重要的数据结构，它有很多不同的应用。从比较简单的应用到复杂的应用，随处可见栈的影子，例如字符串反转、函数活动记录的维护等。理解栈的概念非常重要，因为本书后续的许多其他数据结构和算法都需要用到栈来辅助实现。

栈是一种基本的线性数据结构，它是一种特殊的线性表。它的特点在于仅仅提供对于最新被加入的数据的访问操作，而对于其他元素的访问则加以限制。在栈中，元素的删除和添加只能在表的一端进行。栈中元素的添加和删除操作遵循后进先出（Last In First Out，简称 LIFO）的原则，即后加入栈的数据总是先被访问，而先加入栈的数据总是后被访问。换句话说，下一个能够从栈中被删除的元素永远只能是最近被添加进来的那个。我们总是可以向栈中添加更多的元素，但每当进行一次添加操作时，最近被添加进来的元素随即就变成了可以被最先删除的元素。

下面来看看栈具体是如何工作的，假设有如图 5-2 所示的一个栈，并使用索引 tos 来指向栈顶（top of stack）元素。若 tos 为–1，那么就意味着这个栈是个空栈，如图 5-2(*a*)所示。

- 当向栈中添加一个元素时，先把 tos 的值加 1，然后再向此时 tos 指向的位置添加元素。
- 当删除栈中的元素时，先把 tos 指向的元素从栈中去掉，然后令 tos 的值自减 1。

在栈中添加元素的操作通常被称为入栈（PUSH），从栈中删除元素的操作通常被称为出栈（POP），实现出栈的函数不需要带参数，因为在某一时刻，能够被删除的元素至多只有一个，且它一定位于栈顶。在图 5-2 中首先显示了一个空栈，然后顺序向栈中添加两个元素a与b，最后再把b从栈中删除。

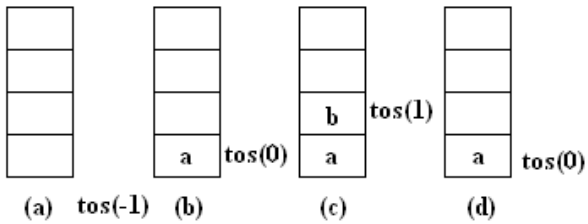


图 5-2　栈

从图 5-2 中可以看出，栈很容易被想象成一种竖立的结构。当向栈中添加一个元素，这个元素会把栈中其他的元素“遮盖”起来。这样，这个新加入的元素便成了栈中唯一可以被访问的元素，同时也是栈中唯一可以被删除的元素。栈的这种性质使栈与链表不同：在栈中可访问的元素是被限定的，不可能做到同时访问栈中的所有元素；而在链表中所有的元素均可被访问。如果想要遍历一个栈，只能够先访问位于栈顶的元素，访问结束后把这个元素从栈中删除，然后再访问新的栈顶元素，如此循环下去，直到遍历结束。由此可以看出，经过遍历操作后的栈将被清空。由于在栈中访问元素是受到限制的，所以相对于链表而言，对栈进行的操作较少。关于栈中操作的实现将在 5.1.2 节中讨论。

5.1.2　栈的操作及实现

栈有两种实现方式，分别为用数组实现与用链表实现。用数组实现的栈称为顺序栈，它在内存中是连续存放的；用链表实现的栈称为链式栈，它在内存中是分散存放的。

由顺序栈的实现方式可知，顺序栈中存放元素的总数是有限的。这样，在向顺序栈中添加元素之前先要判定栈是否已满，如果栈已满则忽略入栈操作。同样地在删除栈中的元素之前应该判断栈是否为空。如果是空栈，则忽略出栈操作。

下面给出顺序栈的类定义。

求职面经点拨

请在完成本部分内容学习后，试回答附录题库中的第 23 题。

```
template <class T>
class ArrayStack{

    int size;
    int tos; //栈顶索引。
    T* contain;

public:
    ArrayStack():size(0),tos(-1),contain(NULL){}
    ArrayStack(int MaxSize);
    void Push(T& element);
    T& GetTop();
    T& Pop();
    bool IsEmpty();
    void MakeEmpty();
};
```

下面给出类中各成员函数实现的源代码。

```
template<class T>
ArrayStack<T>::ArrayStack(int MaxSize)
{//构造函数
    size = MaxSize;
    tos = -1; //初始化 tos 于栈底处。
    contain = new T[size];
}

template<class T>
void ArrayStack<T>::Push(T& element)
{//向栈中添加新元素
    assert(tos != size-1);
    contain[++tos] = element;
}

template<class T>
T& ArrayStack<T>::Pop()
{//从栈中删除元素
    assert(tos!=-1);
    return contain[tos--];
}

template<class T>
T& ArrayStack<T>::GetTop()
{//返回栈顶元素
    assert(tos!=-1);
    return contain[tos];
```

```
}

template<class T>
void ArrayStack<T>::MakeEmpty()
{//置空栈
    tos = -1;
}

template<class T>
bool ArrayStack<T>::IsEmpty()
{//判断栈是否为空。
    return tos==-1;
}
```

请注意下面的语句。语句 assert()是一种断言语句, 功能是对程序进行诊断。如果 assert()语句括号内的条件满足，则程序继续运行下去；如果不满足，则程序终止。正确地运用 assert()可以方便地发现程序中的错误。

```
assert(tos != size-1);
```

由于使用数组，顺序栈中各种方法的时间复杂度均为$O(1)$，由此可见顺序栈是一种实现效率很高的数据结构，但它的不足之处在于对内存的利用不够灵活。下面介绍的链式栈将有效地弥补顺序栈在内存使用方面的不足。

链式栈是利用链表实现的，它在内存中的存放是不连续的，而且存放的节点数目也可以动态地改变，可以根据实际情况进行插入与删除。这种性质使得链式栈可以更合理地利用内存资源，采用链式栈会使程序更加灵活，并且使内存开销更小。链式栈的结构如图 5-3 所示。

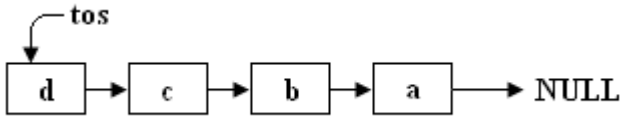


图 5-3　链式栈的结构

在向链式栈中插入新节点时，首先调用 new 语句创建一个节点，并使这个节点指向栈顶节点, 最后更新 tos, 使其指向这个新插入的节点。如果要删除链式栈中的一个节点，只需向栈底方向移动 tos，然后调用 delete 语句完成对原栈顶节点的删除。但与顺序栈不同的是，这里的 tos 并不是一个索引，而是指向栈顶节点的指针。

虽然链式栈是利用链表实现的，但是它与链表类还存在着很多差别。首先，链式栈没有表头节点，而链表类为了统一对空表与非空表的操作引入了表头节点；其次，链式栈没有表头指针与表尾指针, 而链表类中用这两个指针标识链表的开始与结束 ;最后，链表类中的很多方法并不适用于链式栈，这同样是因为栈中的元素是受限制的，用户仅仅可以访问栈顶节点。

下面给出链式栈的类定义。

```
template <class T>
class LinkStackNode{
```

```
public :
    T data;
    LinkStackNode<T>* link;
    LinkStackNode(T& value):link(NULL),data(value){}
};

template <class T>
class LinkStack{

    LinkStackNode<T>* tos;

public :
    LinkStack():tos(NULL){}
    void Push(T& value);
    T& GetTop();
    T Pop();
    bool IsEmpty();
    void MakeEmpty();
};
```

下面给出类中各成员函数实现的源代码。

```
template<class T>
void LinkStack<T>::Push(T& value)
{
    LinkStackNode<T>* add = new LinkStackNode<T>(value);
    add->link = tos;          //节点链入栈。
    tos = add;                //更新栈顶。
}

template<class T>
T& LinkStack<T>::GetTop() {
    return tos->data;
}

template<class T>
T LinkStack<T>::Pop()
{
    assert(tos!=NULL);
    LinkStackNode<T>* old = tos;
    tos = tos->link;           //移动栈顶
    T data = old->data;        //记录原栈顶数据
    delete old;
    return data;
}

template<class T>
bool LinkStack<T>::IsEmpty(){
    return tos == NULL;
}

template<class T>
void LinkStack<T>::MakeEmpty()
{
    while(!this->IsEmpty()){
        this->Pop();
    }
}
```

函数 Pop 的实现需要两个指针相互配合。其中指针 old 用来记载当前的栈顶元素，当

栈顶指针 tos 向栈底方向移动一个节点后，调用 delete 语句删除 old 指向的节点以防止内存泄漏。

经典问题求解

括号匹配问题。

5.1.3 括号匹配问题

程序设计语言是用来编写程序的工具。计算机硬件可以识别的语言是机器语言，通常人们编写的计算机程序所用的高级语言必须被转换成机器语言才可以被计算机理解。将高级语言编写的源程序翻译为等价的计算机可以理解的程序的过程称为编译，而实现编译功能的程序称为编译程序。编译程序是计算机中非常复杂的一种系统程序。在通常的集成开发环境中，编译程序都是最精细、最复杂，也最重要的部分。

一个典型的编译程序一般都含有 8 个部分，即词法分析、语法分析、语义分析、中间代码生成、代码优化程序、目标代码生成、错误检查和处理以及各种信息表格的管理。它们相互协同配合完成编译任务。其中在语法分析阶段，语法分析程序以词法分析程序所输出的用内部编码格式表示的单词序列作为输入，其任务是分析源程序的结构，判断它是否是相应程序设计语言中的一个合法程序，即判定程序的合法性。

现在要求编写一个程序，该程序能够对输入的一段字符串进行括号匹配检查。若语句中左括号与右括号的数目相等，且能够完整地匹配成对，即表示本语句不存在括号匹配上的错误，此时输出“ OK ！”字样。否则表示语句中存在语法错误，程序输出“ Wrong ！”字样。

下面几条语句是存在括号匹配错误的例子：

)(

((()

下面几条语句是不存在括号匹配错误的例子：

( )

$(a)((b)((c)(d)))$

下面给出程序实现的源代码。

```
#include "stdafx.h"
#include <iostream>
#include "LinkStack.h"

using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
    cout<<"请输入括号序列(以0结束):";
    LinkStack<char> small;
    char a;
    do{
        cin>>a;
        switch(a){
        case '(':small.Push(a);break;
```

```
        case ')':
            if(!small.IsEmpty()){
                small.Pop();
                break;
            }
            if(small.IsEmpty()){
                cout<<"Wrong!"<<endl;
                system("PAUSE");
                return 0;
            }
        }
    }while(a!='0');
    if(small.IsEmpty())
        cout<<"OK!"<<endl;
    else
        cout<<"Wrong!"<<endl;

    system("PAUSE");
    return 0;
}
```

完成编码后，编译并运行程序即可。

5.1.4　停车场模拟问题

经典问题求解

停车场模拟问题，论如何成为一名优秀的停车场管理员。

停车场管理员的工作通常是帮助车主将车停放在停车场中或是帮助车主将他们的车开出停车场。停车场中能够停放的车辆的数目很多，这也使得将某辆车从停车场中开走变得复杂。考虑这种情况，假设某人想把他停在停车场中的车开走，但是这辆车却位于一列顺次停放的车的后面，那么，在这种情况下，停车场的管理员需要把这辆车“ 前面的 ”车全部从停车场中暂时清出，然后待这辆车开出后再把那些车重新放入停车场。当然，此时由于空出一个位置，这个位置就由后面的车占据。你的任务就是编程模拟这一情况。这里假设停车场由一个停车甬道组成，且这个甬道最多可停 5 辆车。

我们使用一个名为 data.txt 的文件来记录了某一时段内，该停车场中车辆到来与离开的记录（初始时，停车场为空），其中大写字母A～P是车辆的代号，arrives 表示车辆到来，departs 表示车辆离开。程序需要从该文件中读取这些数据，并基于这些数据模拟停车场的车辆调度情况。这个文件的样例可以从本书的在线支持资源中下载得到。

你可能需要两个栈来完成本题。一个栈用来模拟停车甬道，另一个栈用来暂存从停车场清出的汽车。程序的输出包括每辆车的代码，以及这些代码所表示的车被移动的次数。如果当某辆车到来时，停车场已满，则输出“ 停车场已满 ”消息。最后输出停车场中剩余的车辆的代码。以上面给出的数据作为程序的输入，那么程序的输出结果应当如图 5-4 所示。

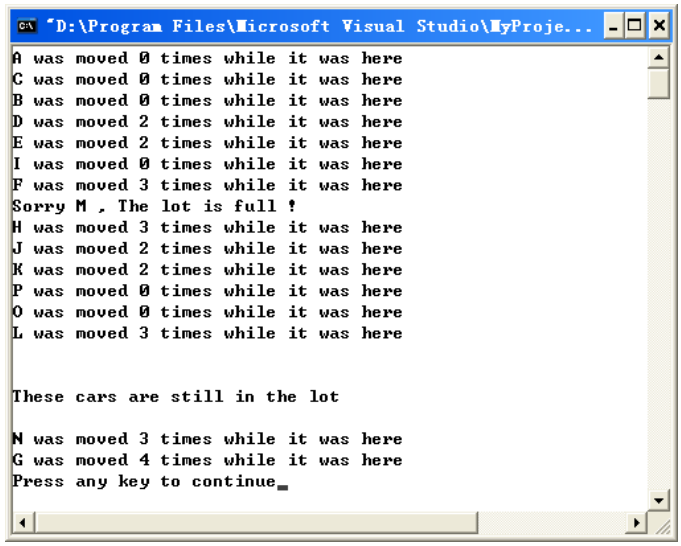

"D:\Program Files\Microsoft Visual Studio\MyProje...
A was moved 0 times while it was here
C was moved 0 times while it was here
B was moved 0 times while it was here
D was moved 2 times while it was here
E was moved 2 times while it was here
I was moved 0 times while it was here
F was moved 3 times while it was here
Sorry M , The lot is full !
H was moved 3 times while it was here
J was moved 2 times while it was here
K was moved 2 times while it was here
P was moved 0 times while it was here
O was moved 0 times while it was here
L was moved 3 times while it was here
These cars are still in the lot
N was moved 3 times while it was here
G was moved 4 times while it was here
Press any key to continue

图 5-4　程序的输出结果

下面给出 car.h 文件的源代码，其中包含了 car 类的定义，每个 car 的实例都拥有两个属性，即代号（用以标识和区别车辆）和被移动的次数。

```
#ifndef CAR_H
#define CAR_H

#include <iostream>
#include <string>

using namespace std;

class car {

private:
    string license;
    int movedTimes;
public:
    car(string,int);
    string getLicense()const;
    int getMovedTimes()const;
    void move();
    virtual ~car();

};

#endif
```

下面给出 car.cpp 文件的源代码。它实现了 car 类中的方法。

```
#include "car.h"

using namespace std;

//构造函数
car::car(string license,int):movedTimes(0),license(license){}
//析构函数
car::~car(){}
```

```
//返回车辆被移动的次数
int car::getMovedTimes()const
{
    return movedTimes;
}
//返回车辆的代号
string car::getLicense()const
{
    return license;
}
//当车被移动时，属性 movedTimes 自加 1
void car::move()
{
    movedTimes++;
}
```

现在给出主程序文件的源代码。其中使用了 STL 中的 stack 容器，关于 stack 容器的具体用法请见本章的后续内容。有兴趣的读者也可以尝试利用我们先前自行实现的 stack 类来改写下面这个程序。

```
#include <string>
#include <stack>
#include <iostream>
#include <fstream>
#include "car.h"

using namespace std;

int main()
{
    string in_fileName = "data.txt";
    ifstream infs(in_fileName.c_str());
    if(!infs){
        cerr<<"Can't open the file "<<in_fileName<<endl;
        return EXIT_FAILURE;
    }

    stack<car*> parking_lot, tempStack;
    car *pcar;
    string license_plate,action;

    //按行读取数据文件
    while(!infs.eof())
    {
        infs>>license_plate>>action;
        if(action=="arrives")
        {
            if(parking_lot.size()<5)
            {
                pcar=new car(license_plate,0);
                parking_lot.push(pcar);
            }
            //当停车场已满时，对于新来车辆输出车场已满的信息
            else
                cout<<"Sorry "<<license_plate<<" ,
                    The lot is full ! "<<endl;
        }
        //对于每个离开的车辆，将相应的元素从栈中删除
```

```
        else if(action=="departs")
        {
            while(!parking_lot.empty()&&
                parking_lot.top()->getLicense() != license_plate)
            {
                tempStack.push(parking_lot.top());
                parking_lot.top()->move();
                parking_lot.pop();
            }
            //输出每辆车的移动情况
            if(parking_lot.top()->getLicense() == license_plate)
            {
                cout<<parking_lot.top()->getLicense()<<"
                    was moved "<<parking_lot.top()->getMovedTimes()<<"
                    times while it was here"<<endl;
                delete parking_lot.top();
                parking_lot.pop();
            }
            else
                cout<<"Exception!"<<endl;
            while(!tempStack.empty())
            {
                parking_lot.push(tempStack.top());
                tempStack.pop();
            }
        }
    }
    //输出仍然被停在停车场中的车辆
    cout<<"\n\nThese cars are still in the lot \n\n";
    while(!parking_lot.empty())
    {
        cout<<parking_lot.top()->getLicense()<<"
        was moved "<<parking_lot.top()->getMovedTimes()<<"
        times while it was here"<<endl;
        delete parking_lot.top();
        parking_lot.pop();
    }

    infs.close();

    return EXIT_SUCCESS;
}
```

完成编码后，编译并运行程序即可。

5.2 排队的智慧

队列经常被拿来和栈进行比较。队列也是一种基本的线性数据结构，但操作特性却与栈结构截然相反。队列中，元素总是在一端加入，而从另一端移出，也就是说最新进入队列的元素将最后被移出。

5.2.1 队列的结构

队列是一种基本的线性数据结构，其特点是只允许在结构的一端对元素进行添加操作，

而元素的删除操作只能在结构的另一端进行。通常将进行元素添加操作的一端称作队列的尾，而将进行元素删除操作的一端称作队列的头。队列的这种一端删除、一端添加的机制是通过一种被称为先进先出（FIFO，First In First Out）的原则来实现的。先进先出意味着最先被加入到队列中的数据也将被最先从队列中删除。换言之，如果想要向队列中添加元素，那么这个操作仅能在队列的尾部进行。如果想要从队列中删除一个元素，那么这个操作仅能在队列的头部进行。

问题导入

想想在电影院排队买票时的场景。

在现实世界中存在很多队列的实例。例如，在电影院的售票窗口排队等待买票的人，就是通过队列这种数据结构组织在一起的。人们按先来后到的顺序排成一队，最先买到票的人就是最先来的人。当某人买完票，他就会从队列的最前端离开，这就相当于删除操作。而添加的操作却仅能在队列的尾部进行，因此新来的人就只能排在队列的最后。此外还有像公共汽车站人们排队等车、医院中病人们排队候诊等都是现实生活中队列的例子。

队列在计算机科学和软件开发中有着很多不同的应用。队列可以被用来提供缓冲区。缓冲区就是一块用来进行临时信息存储的区域，这里的数据稍后就会被处理，因此仅仅是暂时存在缓冲区里。例如，网络中的路由器都提供有缓冲区，缓冲区将收到的数据包组织在一个队列中，以保证最先收到的数据包会被最先发出去。读者一定有过在线收看视频节目的经历，通常与互联网相连接的流媒体程序在播放视频片段时，会将收到的数据包维护在一个队列中，然后按顺序将其播出，这种缓冲机制可以保证当网络状况不稳定时视频播放的流畅性。

操作系统也使用队列来维护作业调度，管理输入/输出。在维护作业调度时，如果有很多等待处理的程序，操作系统就会通过队列来决定下一个将要被运行的程序。此外，在打印机中，如果待处理的打印任务很多，那么它们也将被存储在队列中，并由此来决定下一个要执行的打印任务是哪个。这种策略也被称为先来先服务（FCFS，First Come First Served）。

在计算机上可以有很多不同的方法来实现队列，其中最简单的方式莫过于利用数组。数组开头的位置（下标为 0）存储队列中的第 1 个元素，其他的元素依次存储。当要向队列中添加新元素时，直接把这个元素添加到数组的末尾即可，这个操作类似于向栈中添加一个新元素。但当从队列中删除一个元素时，则只能删除队列中位于“最前面”的元素，这个操作与从栈中删除一个元素正好相反。

队列与栈的另一个不同之处在于队列需要两个索引，即队头（front）和队尾（back），而栈只需要一个索引值。当添加元素时，先把新元素添加到此时 back 指示的位置（在此之前要判断队列是否已满），然后令索引 back 向后移动一个单位；当删除元素时，直接令 front 向后移动一个单位即可。队列的结构如图 5-5 所示。

图 5-5 中首先创建一个空队列，然后依次向队列中添加两个元素a与b。这时，a存储于队列的头部，是队列中的第 1 个元素，b存储于与a相邻的下一单元中，是队列中的第 2 个元素。在向队列中添加元素时，索引 back 会向后移动，而索引 front 不会改变。

当从队列中删除元素时，先删除位于队列头部的a，随后删除b。这时，索引 front 会向后移动，而索引 back 不会改变。

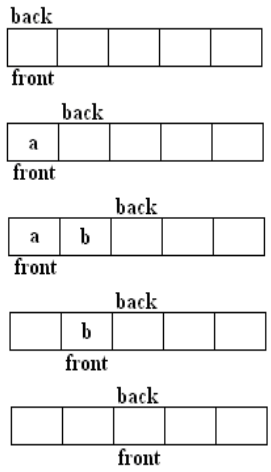


图 5-5　队列的基本结构

队列中元素的访问是受到限制的，用户只能访问位于队列头部的元素，而无法访问位于其他位置的元素。同栈类似，对队列进行遍历将会把队列置空。

队列的基本操作有：向队尾添加一个新元素、删除位于队列头部的元素、获得当前队头元素的值、置空以及判空等。

5.2.2　队列的操作及实现

队列的实现，可以使用数组，也可以使用链表。用数组实现的队列称为循环队列，用链表实现的队列称为链式队列。

在向用数组实现的队列中添加新元素之前要判断当前队列是否已满，以防止产生溢出错误。但是，这种“溢出”并不是真正的溢出，因为在数组的前端可能还有空位置，所以这是一种假溢出。为了充分地利用内存资源可以把数组的首尾连接起来，当发生“溢出”时，把元素添加到数组的头部，这样便形成一个循环队列。不过，循环队列只是在逻辑上形成一个环，在实际的内存中它仍存储在连续的线性空间内。

下面给出循环队列的定义。

```
template <class T>
class CirQueue{
    int size;
    int front;
    int back;
    T* contain;

public:
    CirQueue():size(0),front(0),back(0),contain(NULL){}
    CirQueue(int MaxSize);
    void EnQueue(T& element);
```

```
    T DelQueue();
    T& GetFront();
    void MakeEmpty();
    bool IsEmpty();
    bool IsFull();
};
```

下面给出类中各成员函数实现方法的源代码。

```
template <class T>
CirQueue<T>::CirQueue(int MaxSize):size(MaxSize+1),front(0),back(0)
{//构造函数
    contain = new T[MaxSize+1];
}

template <class T>
void CirQueue<T>::EnQueue(T& element)
{//向队列中添加新元素
    assert(!IsFull());
    contain[back] = element;
    back = (back+1)%size;
}

template <class T>
T CirQueue<T>::DelQueue()
{//从队列中删除元素
    int old = front;
    front = (front+1)%size;
    return contain[old];
}

template <class T>
T& CirQueue<T>::GetFront()
{//返回队头元素
    assert(!IsEmpty());
    return contain[front];
}

template <class T>
void CirQueue<T>::MakeEmpty()
{//置空队列
    front = back = 0;
}

template <class T>
bool CirQueue<T>::IsEmpty()
{//判断队列是否为空
    return front == back;
}

template <class T>
bool CirQueue<T>::IsFull()
{//判断队列是否已满
    return (back+1)%size == front;
}
```

构造函数在创建队列时多开辟了一个存储空间。这是因为在循环队列中索引 back 总是指向最后一个元素的后继（见图 5-3），而这个位置不存放任何元素。如果希望队列最

多存放 5 个元素，那么构造函数实际上创建了长度为 6 的数组，其中 5 个位置用于存放数据，余下一个位置什么元素也不放，当队列中元素已满时，back 便指向这个位置，如图 5-6 所示。

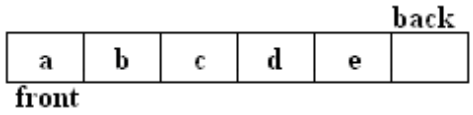


图 5-6　满队列

由于循环队列在逻辑上呈现出环状，所以在添加新元素前需要调用语句：

```
back= (back+1)%size;
```

该语句的作用是使到达数组尾部的索引 back 重新回到数组头部。如果此时 back 指示的位置没有存储元素便把新元素添加到这个位置；如果此位置已储存了元素则表明队列已满，忽略添加操作。

队列的判空比较简单，只要当前队列中索引 front 等于 back，则意味着队列中元素个数为零，即队列为空队列。如果当前队列中索引 front 与 back 满足条件：

```
(back+1)%size == front;
```

则意味着索引 back 的后继为 front，即队列为满队列。

与栈相同，队列除了可以用数组实现以外还可以用链表实现。利用链表实现的队列称为链式队列。相对于链式栈而言，链式队列更接近于链表的实际结构。链式队列的结构见图 5-7。

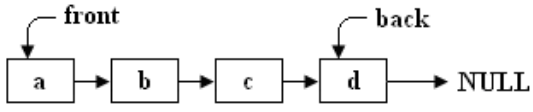


图 5-7　链式队列的结构

下面给出链式队列的类定义。

```
template <class T>

class LinkQueue{

    struct LinkQueueNode
    {
        T   data;
        LinkQueueNode *link;
        LinkQueueNode(T & theData, LinkQueueNode * n = NULL)
            : data(theData), link(n) { }
    };
    LinkQueueNode* front;
    LinkQueueNode* back;

public:
    LinkQueue();
    ~LinkQueue();
```

```
    void EnQueue(T& element);
    T DelQueue();
    T& GetFront();
    void MakeEmpty();
    bool IsEmpty();
};
```

链式队列的入队操作和出队操作在具体实现上遵循了第 4 章中关于链表中插入或者删除节点的基本方法。然而，链式队列的插入操作只能在链表的尾部进行，其删除操作只能在链表的头部进行。如图 5-8 所示，在队尾插入一个节点时，需要把原链表最后一个节点中所持有的空指针指向新插入的尾节点，同时将指向原尾节点的 back 指针指向新插入的尾节点。在队头删除一个节点时，则需要将指向原头节点的 front 指针指向新的头节点，即原头节点的后继节点。

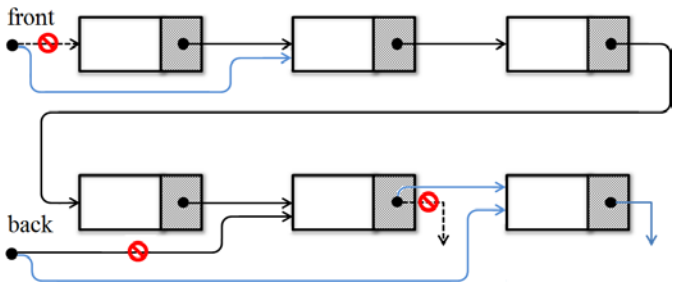


图 5-8　链式队列的操作

下面给出类中各成员函数实现的源代码。

```
template <class T>

LinkQueue<T>::LinkQueue()
{
    front = back = NULL;
}

template <class T>
LinkQueue<T>::~LinkQueue()
{
    this->MakeEmpty();
}

template <class T>
void LinkQueue<T>::EnQueue(T& value)
{
    if(this->IsEmpty())
        front = back = new LinkQueueNode(value);
    else
        back = back->link = new LinkQueueNode(value);
}

template <class T>
T LinkQueue<T>::DelQueue()
{
    LinkQueueNode* old = front;
    T data = old->data;
    front = front->link;
```

```
    delete old;
    return data;
}

template <class T>
T& LinkQueue<T>::GetFront()
{
    assert(!IsEmpty());
    return front->data;
}

template <class T>
void LinkQueue<T>::MakeEmpty()
{
    while(!this->IsEmpty()){
        this->DelQueue();
    }
}

template <class T>
bool  LinkQueue<T>::IsEmpty()
{
    return front == NULL;
}
#endif
```

经典问题求解

舞伴问题。

5.2.3 舞伴问题

假设在周末舞会上，男士们和女士们进入舞厅时，各自排成一队。跳舞开始时，依次从男队和女队的队头各出一人配成舞伴。若两队初始人数不相同，则较长的那一队中未配对者等待下一轮舞曲。现要求写一算法模拟上述舞伴配对问题。

舞伴问题是使用队列进行模拟的典型问题。编程实现时，先建立两个队列 Mdancer 与 Fdancer，分别用来存放男、女舞伴。接下来向队列中输入到达舞会的实际人数：当男舞伴多于女舞伴时，Mdancer 将长于 Fdancer。反之亦然。

当全部舞伴入队完毕时，程序开始输出配对情况。每次按照先来后到的顺序从队列头部分别取出一个男舞伴与一个女舞伴进行配对。如果最后两个队列全部为空，则说明没有人剩下，全部舞伴均能配对；若其中一支队列为空而另外一支队列非空，则非空那支队列中的舞伴在这轮舞会中落单。此时程序输出非空队列第 1 个人的姓名，表示下一个被配对的舞伴将是这个人。

下面给出程序实现的源代码。

```
#include "stdafx.h"
#include "LinkQueue.h"
#include <string>
#include <iostream>

using namespace std;

struct dancer{
    string name;
    char sex;
};
```

```
int _tmain(int argc, _TCHAR* argv[])
{
    cout<<"请输入舞者总人数:";
    int num;
    cin>>num;
    LinkQueue<dancer> Mdancer;
    LinkQueue<dancer> Fdancer;
    for(int i=0;i<num;i++){
        cout<<"请输入舞者性别(f or m)及姓名:";
        char sex;
        cin>>sex;
        string name;
        cin>>name;
        dancer newdancer;
        newdancer.name = name;
        newdancer.sex = sex;
        if(sex == 'f')
            Fdancer.EnQueue(newdancer);
        if(sex == 'm')
            Mdancer.EnQueue(newdancer);
    }

    while(!Mdancer.IsEmpty() && !Fdancer.IsEmpty()){
        cout<<Mdancer.DelQueue().name<<"\t<---->\t"
            <<Fdancer.DelQueue().name<<endl;
    }

    if(!Mdancer.IsEmpty()){
        cout<<"Mr. "<<Mdancer.GetFront().name<<" is waiting!"<<endl;
    }
    else if(!Fdancer.IsEmpty()){
        cout<<"Ms. "<<Fdancer.GetFront().name<<" is waiting!"<<endl;
    }
    else
        cout<<"OK!"<<endl;

    system("PAUSE");
    return 0;
}
```

完成编码后，编译并运行程序即可。

5.2.4　杨辉三角问题

经典问题求解

杨辉三角问题，你是否可以对程序稍做调整，让其输出一个完整的三角，而非仅仅是最后一行?

计算牛顿二项式系数的一个有效的方法是利用杨辉三角。杨辉三角的构造方式是将三角形每一行两边的元素置为 1，其他元素为这个元素“肩”上两元素之和。该三角曾经在我国宋朝数学家杨辉 1216 年所著的《详解九章算法》中出现过。杨辉当时摘录的是北宋时期数学家贾宪的《开方作法本源》一书中的“开方作法本源图”。因而，人们把这个表又称为贾宪三角。在欧洲这个表被称为帕斯卡三角形。帕斯卡是 1654 年发现这一规律的，比杨辉迟 393 年，比贾宪迟 600 年。一个简单的五阶杨辉三角如下所示。

```
                1
            1       1
        1       2       1
    1       3       3       1
1       4       6       4       1
```

现在请利用队列实现杨辉三角的构造。输入杨辉三角的行号，程序打印相应行的二项式系数。为此，先建立一个队列并初始化队列中的元素为 1，1 这个队列将用来迭代生成任意行的牛顿二项式系数。接下来根据用户输入的行号，程序决定循环次数。在这些循环中，程序根据杨辉三角实际的构造情况模拟构造过程：每次形成一个新的二项式系数序列，并将这一序列保存在一个新的队列中。本次循环结束时，这个新构造的序列将作为下次循环构造另一个二项式序列的参照序列。由于 LinkQueue 类中并没有提供链表类的赋值方式，所以程序中要另外编写代码实现队列的赋值功能。

下面给出程序实现的源代码。

```
#include "stdafx.h"
#include "LinkQueue.h"
#include <iostream>

using namespace std;

template<class T>
void evaluate(LinkQueue<T>& ori,LinkQueue<T>& target){
    ori.MakeEmpty();
    while(!target.IsEmpty()){
        int tmp_value = target.DelQueue();
        ori.EnQueue(tmp_value);
    }
}

int _tmain(int argc, _TCHAR* argv[])
{
    cout<<"请输入杨辉三角阶数 i(i>2):";
    int num;
    cin>>num;
    LinkQueue<int> ori;

    int ini_value = 1;
    ori.EnQueue(ini_value);
    ori.EnQueue(ini_value);
    LinkQueue<int> next;
    for(int i=0;i<num-2;i++){
        next.EnQueue(ini_value);
        while(!ori.IsEmpty()){
            int i=ori.DelQueue();
            if(!ori.IsEmpty())
            {
                int tmp = i+ori.GetFront();
                next.EnQueue(tmp);
            }
            if(ori.IsEmpty())
                next.EnQueue(i);
        }
        evaluate(ori,next);
    }
    cout<<"杨辉三角第"<<num<<"行内容如下:"<<endl;
```

```
    while(!ori.IsEmpty()){
        cout<<ori.DelQueue()<<" ";
    }
    cout<<endl;

    system("PAUSE");
    return 0;
}
```

完成编码后，编译并运行程序即可。

5.2.5　游程编码问题

经典问题求解

游程编码问题，尽管这是一种目前很少使用的压缩技术，但在数字图像编解码领域它仍然有被用到。

游程编码（RLE，Run Length Encoding）是一种相对简单的无损压缩编码技术，它在早期的图像压缩中被广泛使用，PCX 就是一种使用游程编码方式的图像格式。它的原理很简单，即基于图像的统计特性基础，将一个相同值的连续串用一个代表值和串长来代替。如黑或白像素点连续出现的像素点数称为游程长度，简称长度。该方法更适合于二值图像或黑白图像的编码压缩。这主要是因为它们的分布都属于平稳的随机分布，在同一行或相邻行的像素之间具有较强的相关性。

这种编码方式应用在一幅具有连续相同的大面积颜色块图像时，压缩效果十分可观。然而，不幸的是这种情况在实际中可能并不多见。特别地，当一幅图像中每两个相邻点的颜色都不同时，采用游程编码非但不能对图像数据进行压缩，反而会使数据量增加。也就是通常所说的 RLE 中的负压缩现象。例如有如下报文：MyDogHasFleas，那么它的游程编码就可以记作：1M1y1D1o1g1H1a1s1F1l1e1a1s。经过 RLE 编码之后，报文的长度膨胀了一倍。这也是该算法的一个致命弱点。

请编程实现二值序列的游程编码。要求输入一段二值（只包含 0 和 1）序列，然后程序输出它的游程编码。例如，有如下一段报文 00011100110101000011，经过编码后它的游程编码就可以记作：3322111142。下面给出程序实现的源代码，代码中使用到了 STL 中的 queue 容器，关于它的用法请参见本章后续的内容。

```
#include "stdafx.h"
#include <queue>
#include <iostream>

using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
    queue<int> picture;
    char a;
    float pow = 0;
    float oriSize;
    cout<<"请输入 01 序列(以#结束):";
    do{
        cin>>a;
        if(a!='#')
            picture.push(a);
    }while(a != '#');

    char pic = picture.front();
```

```
    picture.pop();
    int size = picture.size()+1;
    oriSize = size;

    cout<<"编码后:";
    while(!picture.empty()){
        if(pic == picture.front()){
            pic = picture.front();
            picture.pop();
        }
        else{
            cout<<size - picture.size();
            pow++;
            size = picture.size();
            pic = picture.front();
            picture.pop();
        }
    }
    cout<<size<<endl;
    pow++;
    cout<<"压缩率为:"<<pow/oriSize<<endl;

    system("PAUSE");
    return 0;
}
```

完成编码后，编译并运行程序即可。

5.3 优先级队列——兼谈页面置换算法

在 5.2 节介绍的队列中，元素与元素之间不存在优先级次序。也就是说队列的实现严格按照 FIFO 的原则，每次从队列中取出的是最早加入其中的元素。但在实际应用中，队列中的元素可能需要一定的优先级，每次从队列中取出具有特定优先级的元素。这种队列叫作优先级队列。本节将介绍这种特殊队列的结构及实现。

5.3.1 优先级队列的结构

先来看一个现实当中的例子。5.2 节曾经介绍过打印机待处理的打印任务是存储在队列中的，但在某些时候这也许并不能很好地解决实际问题。例如，假设某公司仅有一台可以工作的打印机，且这台打印机既要打印来自经理的文件又要打印来自普通员工的文件。如果经理的文件来得比普通员工的文件晚，那么经理就要等待了，而通常经理的文件总是比普通员工的文件重要。显然此时使用普通队列就不合适了。

可以通过把打印机中的待打印任务组织在优先级队列中的方式来解决这个问题。打印机每次根据一定的优先级次序从队列中取出文件。这样一来，即使经理的文件来得较晚，如果它有足够高的优先级，那么打印机也会优先处理这个文件。当处理完来自经理的文件后，打印机取出具有次高优先级的打印任务，这个任务可能来自副经理，也可能来自某个员工，总之这个文件是没有经理的文件重要的。利用优先级队列工作的打印机总是优先打印最急迫的文件，然后再打印其他文件，这样才不会耽误公司的重

要事宜。如果若干个任务的优先级相同，那么打印机将按照“先来先服务”的顺序完成它们。

计算机中也有很多地方会用到优先级队列。例如，操作系统在进行 I/O 设备的管理时，最常采用的分配算法有两种，即先请求先服务和优先级高者优先两种算法。由于同计算机相连的 I/O 设备可能有很多，当众多外设都向操作系统发出请求时如何进行资源分配呢？当使用先请求先服务算法时，算法把所有发出 I/O 请求的进程，按发出请求的先后顺序排成一个等待该设备的队列。I/O 调度程序把 I/O 设备分配给队列中的第一个进程。如果使用的是优先级高者优先算法，那么进程等待队列就按优先级从高到低排队，一旦 I/O 调度程序确定了满足 I/O 请求的相对顺序之后，如果有可用的资源，I/O 设备分配程序就会优先满足那个具有高优先级的 I/O 请求。显然此时使用优先级队列比较合适。

通过以上举例，我们可以定义如果有一个队列满足每次从队列中取出的元素总是具有最高优先级的元素这样的条件，那么它就是一个优先级队列。特别地，如果从优先级队列中删除一个元素时出现多个元素具有相同优先级的情况，那么就把具有相同优先级的元素看作是一个普通队列，并根据先进先出的原则进行操作。

在操作系统中有很多情况下都会使用 FIFO 策略和优先级高者优先策略。除了前面提到的 I/O 设备管理以外，还有在作业调度、页面置换和磁盘调度中都会使用这些策略。在本节的末尾，我们简单介绍一些比较常用的算法，以深化读者对于此部分内容的理解。

操作系统在进行存储管理时，如果在地址映射过程中，页表中发现所要访问的页不在内存，就会产生缺页中断。操作系统接到此中断信号后，就调出缺页中断处理程序，根据页表中给出的外存地址，将该页调入内存，使作业继续运行下去。如果内存中有空闲块，则分配一页，将新调入页装入内存，并修改页表中相应页表项目的驻留位及相应的内存块号。若此时内存中没有空闲块，则要淘汰某页，若该页在内存期间被修改过，则要将其写回外存。在进行页面淘汰时，如何选定要淘汰的页面呢？通常可用的算法有先进先出页面淘汰算法、最近最少使用页面淘汰算法和最不经常使用页面淘汰算法等。

小提示

操作系统中的页面置换算法（或称页面淘汰算法）是操作系统中的基本问题，属于必知必会的部分，笔试中这类问题也非常普遍。

先进先出页面淘汰算法（FIFO），顾名思义，简言之就是选择在内存中驻留时间最长的页面并淘汰之。由于其特点与队列的属性一致，因此可以使用队列来维护内存中的页面。

最近最少使用页面淘汰算法（LRU），就是选择最后一次访问时间距离当前时间最长的页面并淘汰之，即淘汰没有使用的时间最长的页面。该算法的设计出发点在于认为如果某一页面被访问了，那么它很有可能马上又被访问，反之，如果某一页面很久没有被访问，那么最近也不大可能被访问。

最不经常使用页面淘汰算法（NFU），就是选择访问次数最少的页面并淘汰之。但在实现过程中，该算法需要一个计数器，而且每个页面一个，初值为 0。每次时钟中断时，

计数器自加。发生缺页中断时，选择计数器值最小的页面淘汰。

可见最近最少使用页面淘汰算法和最不经常使用页面淘汰算法都是通过判断队列中元素的优先级来决定淘汰哪一个的。只是两个算法断定优先级的方法不同。

假设某程序在内存中分配 3 个页面，初始为空，页面走向为 4、3、2、1、4、3、5、4、3、2、1、5。求分别使用 FIFO 策略和 LRU 策略时的缺页次数。

通过分析易知，可以使用一个容量为 3 的队列来模拟基于 FIFO 策略的内存页面淘汰过程，该过程如表 5-1 所示。

表 5-1　基于 FIFO 策略的页面置换过程

FIFO	4	3	2	1	4	3	5	4	3	2	1	5
页 1	4	3	2	1	4	3	5	5	5	2	1	1
页 2		4	3	2	1	4	3	3	3	5	2	2
页 3			4	3	2	1	4	4	4	3	5	5
缺页	×	×	×	×	×	×	×	√	√	×	×	√

在表 5-1 中，如果发生缺页中断，即程序所需的内容不在内存页中，系统就从外存中将该页内容调入，如果内存页已满则按照 FIFO 原则替换其中的某页，这时记为发生一次页面置换，用符号“ × ”表示。否则如果程序所需的内容位于内存页中，就不发生页面置换，这时用符号“ √ ”表示。最后求得当使用 FIFO 策略时，共发生缺页中断 9 次。

下面使用一个容量为 3 的优先级队列来模拟基于 LRU 策略的内存页面淘汰过程，并将最后一次访问时间距离当前时间最长的页面赋予最高优先级，该过程如表 5-2 所示。

表 5-2　基于 LRU 策略的页面置换过程

LRU	4	3	2	1	4	3	5	4	3	2	1	5
页 1	4	3	2	1	4	3	5	4	3	2	1	5
页 2		4	3	2	1	4	3	5	4	3	2	1
页 3			4	3	2	1	4	3	5	4	3	2
缺页	×	×	×	×	×	×	×	√	√	×	×	×

在表 5-2 中，如果发生缺页中断，即程序所需的内容不在内存页中，系统就从外存中将该页内容调入，如果内存页已满则按照 LRU 原则替换其中的某页，这时记为发生一次页面置换，用符号“ × ”表示。否则如果程序所需的内容位于内存页中，就不发生页面置换，这时用符号“ √ ”表示。最后求得当使用 LRU 策略时，共发生缺页中断 10 次。

对比表 5-1 和表 5-2 后，需要提示读者注意的是，第 8、第 9 两次页面访问请求时，由于应用不同策略，因而导致内存页中产生差异。在应用队列进行基于 FIFO 策略的页面置换，当发生第 8、第 9 两次页面访问请求时，由于所请求的内容均位于内存中，此时队列中的数据依然是 4 3 5，即并未发生改变。但是在使用 LRU 策略时，因为使

用的是优先级队列，当发生第 8 次页面访问请求时，由于所请求的内容均位于内存中，此时队列中元素的优先级发生了改变，所以导致最近被访问的元素获得了最低的优先级，此时队列中的数据依次是 3 5 4。同理当发生第 9 次页面访问请求时，队列再次调整其中元素的优先级，这时队列中的数据依次是 5 4 3。于是当发生第 10 次页面访问请求时，被置换出去的是优先级最高的元素 5。

举以上两个例子的目的在于帮助读者深化对于队列和优先级队列工作方式的理解。5.3.2 节将介绍有关优先级队列的类定义和操作实现等方面的内容。

5.3.2　优先级队列的实现

实现优先级队列的方法有很多种：可以利用数组实现，也可以利用链表实现，还可以利用堆实现。本节中只介绍利用链表实现的优先级队列。

```
template <class T>
class PriQueueNode{
public:
    T data;
    PriQueueNode<T>* link;
    PriQueueNode(T& value):data(value),link(NULL){}
};

template <class T>
class PriQueue{

    PriQueueNode<T>* front;
    PriQueueNode<T>* back;

public:
    PriQueue():front(NULL),back(NULL){}
    void EnQueue(T& element);
    T DelQueue();
    T& GetFront();
    void MakeEmpty();
    bool IsEmpty();
};

template <class T>
void PriQueue<T>::EnQueue(T& value)
{
    PriQueueNode<T>* add = new PriQueueNode<T>(value);
    if(front == NULL)
        front = back = add;
    else
    {
        back->link = add;
        back = back->link;
    }
}

template <class T>
T PriQueue<T>::DelQueue()
{
    PriQueueNode<T>* seek = front->link;    //遍历队列，寻找最小元素的指针
    PriQueueNode<T>* min = front;           //指向最小元素的指针
```

```
    T minData;                                  //记录最小元素的数据
    while(seek!=NULL){
        if(seek->data < min->data)              //寻找最小元素
            min = seek;
        seek = seek->link;
    }

    seek = front;                               //指针重回队列头部
    while(seek!=NULL&&seek->link!=min)          //寻找最小元素的前驱节点
    {
        seek = seek->link;
    }

    if(seek == NULL){                           //如果最小元素位于队列头部
        minData = front->data;
        front = front->link;                    //更新队头指针
        delete min;
    }

    if(min->link == NULL)                       //如果最小元素位于队列尾部
        back = seek;                            //更新队尾指针

    if(seek!=NULL){                             //如果最小元素位于队列中间，无须更新
        minData = min->data;                    //队头指针与队尾指针，直接删除即可
        seek->link = min->link;
        delete min;
    }
    return  minData;                            //返回数据
}

template<class T>
T& PriQueue<T>::GetFront(){
    assert(!this->IsEmpty());
        return front->data;
}

template<class T>
void PriQueue<T>::MakeEmpty(){
    while(!IsEmpty()){
        this->DelQueue();
    }
}

template<class T>
bool PriQueue<T>::IsEmpty()
{
    return front == NULL;
}
```

由于优先级队列的实现要利用关系运算符“ < ”，所以若队列中存储的是用户自定义的数据类型，则用户需要自行重载运算符“ < ”。

5.4 STL 中的栈与队列

容器是 STL 中最突出的优势。因为它们比数组更强大、更灵活，容器不仅可以动态增

长或缩减，还可以管理属于它们自己的内存，并跟踪它们所拥有的对象数目，甚至提供支持它们的许多复杂操作。通常容器可以分为两类，即序列式容器（Sequence container）和关联式容器（Associated container）。其中，序列式容器是指每个元素都有固定位置的容器，这个“固定的位置”取决于插入时机和地点，和元素值无关。序列式容器包括 vector、deque 以及前面曾经讲过的 list 等。关联式容器是指元素位置取决于特定的排序准则，而和插入顺序无关的容器，典型的关联式容器包括 set、multiset、map 和 multimap 等。本节介绍 STL 中提供的用于模拟栈结构和队列结构的容器。

5.4.1　STL 中的 stack

STL 中的 stack 类可以作为栈来使用。事实上它只是一种适配器，它是通过简单地修饰 deque 的接口而形成的另外一种容器类型。什么是适配器呢？适配器本来（Adaptor）是设计模式中的一种，它意味着将一个类的接口转换为另一个类的接口，使原本因接口不相容而不能合作的类，可以一起运作。STL 基于这种思想也引入了适配器（当然这里不再是一种设计模式）。简单地说，STL 中的适配器是用来修改其他组件接口的 STL 组件，是带有一个参数的类模板（这个参数是操作的值的数据类型）。STL 定义了 3 种形式的适配器 :容器适配器（container adaptor）、迭代器适配器（iterator adaptor）和函数适配器（function adaptor），其中用来改变容器接口的适配器就称作容器适配器。研究 STL 中的 stack 类具体是如何实现在本书中的意义不大，下面重点介绍 stack 类的使用。

stack 类中主要提供了 5 个常用方法，它们是：

- void push(const value_type& x)

将一个新元素压入栈中。

- void pop()

如果栈不为空，弹出栈中的一个元素。

- value_type& top()

返回栈顶元素的引用，如果栈为空，结果未定义，返回的是一个引用。

- size_type size() const

返回栈中元素的个数。

- bool empty() const

判断栈是否为空，若为空返回 true，否则返回 false。

下面给出一段简单的示例代码，它说明了上述函数的使用方法，注意在使用 stack 类之前，需要引入头文件<stack>。

```
#include "stdafx.h"
#include <iostream>
```

```
#include <cstdlib>
#include <stack>

using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
    stack<int> myStack;

    //入栈与出栈
    myStack.push(5);
    myStack.push(6);
    myStack.push(7);
    myStack.pop();

    cout << myStack.top() << endl;
    cout << myStack.size() << endl;
    cout << myStack.empty() << endl;
    system("PAUSE");
    return 0;
}
```

下面这个例子更具一些实际意义。这段示例程序的作用是从一个文本文件中按行读入文本，并将其存储在栈里，然后通过栈来实现文本内容的反转（以行为单位）。即原文的最后一行将成为输出结果的首行，而原文的首行将成为输出结果的末行。

```
#include "stdafx.h"
#include <iostream>
#include <fstream>
#include <string>
#include <cstdlib>
#include <stack>

using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
    ifstream inf;
    inf.open("temp.txt");

    if ( !inf ) {
        cerr << "cannot open file for input!" << endl;
        return EXIT_FAILURE;
    }

    stack<string> s;
    string line;

    while (getline(inf, line)) {
        s.push(line);
    }

    inf.close();

    while (!s.empty()) {
        cout << s.top() << endl;
        s.pop();
    }
```

```
    system("PAUSE");
    return 0;
}
```

完成编码后，编译并运行程序即可。

5.4.2　STL 中的 queue

STL 中的 queue 类可以作为队列来使用。与 stack 相同，它也是一种容器适配器，它是通过简单地修饰 deque 的接口而形成的另外一种容器类型。queue 类中主要提供了 6 个常用方法，它们是：

- void push(const value_type& x)

将一个新元素接到队列的末端。

- void pop()

弹出队列中的第 1 个元素，返回值是 void。

- value_type& front()

存取队列中的第 1 个元素，返回的是一个引用。

- value_type& back()

存取队列中最后一个元素，返回的是一个引用。

- size_type size() const

返回队列中元素的个数。

- bool empty() const

判断队列是否为空，若为空返回 true，否则返回 false。

下面给出一段简单的示例代码，它说明了上述函数的使用方法，注意在使用 queue 类之前，需要引入头文件<queue>。

```
#include "stdafx.h"
#include <iostream>
#include <cstdlib>
#include <queue>

using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
    queue<int> myQueue;

    //添加元素与删除元素
    myQueue.push(5);
    myQueue.push(6);
    myQueue.push(7);
```

```
    myQueue.pop();

    cout << myQueue.front() << endl;
    cout << myQueue.back() << endl;
    cout << myQueue.size() << endl;
    cout << myQueue.empty() << endl;

    system("PAUSE");
    return 0;
}
```

下面这个例子更具一些实际意义。这段示例程序用队列模拟了基于 FIFO 策略的内存页面淘汰过程，它综合运用了前面所提及的各个函数，很具代表性。读者可以将程序的运行结果和表 5-1 对照一下，以便理解该程序模拟的具体过程。

```
#include "stdafx.h"
#include <iostream>
#include <cstdlib>
#include <queue>

using namespace std;

bool existOrNot(int num, queue<int> myQueue)
{
    while(!myQueue.empty())
    {
        if(myQueue.front()==num)
            return true;
        else myQueue.pop();
    }
    return false;
}

int _tmain(int argc, _TCHAR* argv[])
{

   queue<int> memQueue;

    int page[] ={4, 3, 2, 1, 4, 3, 5, 4, 3, 2, 1, 5};
    int times = 12;
    int tmpValue;

    for(int i = 0; i < 12; i++)
    {
        if(memQueue.size()<3)
            memQueue.push(page[i]);
        else
        {

            if(existOrNot(page[i], memQueue))
            {
                times--;
            }
            else
            {
                memQueue.pop();
                memQueue.push(page[i]);
            }
        }
```

```
        int size = memQueue.size();
        while(size>0)
        {
            cout<<memQueue.front()<<" ";
            tmpValue = memQueue.front();
            memQueue.pop();
            memQueue.push(tmpValue);
            size--;
        }

        cout<<endl;
    }

    cout<<"使用 FIFO 策略时，共发生缺页中断"<<times<<"次."<<endl;
    system("PAUSE");
    return 0;
}
```

完成编码后，编译并运行程序即可。

5.4.3　STL 中的 priority_queue

STL 中的 priority_queue 和 queue 都是容器适配器，通过前面的介绍可知，queue 实现的是一个先进先出的队列，priority_queue 实现的是优先级队列。尽管它也是一个队列，但该模板类可以保证每次通过 pop 从它里面取得的元素都是剩下元素中优先级最高的，在每次通过 push 添加元素时，整个序列都会在必要时进行重排。在这个过程中，它使用堆算法保持序列的有序化。此外，还需要说明的是，queue 使用的是 deque 作为内部容器，而 priority_queue 则使用 vector 作为内部容器。

priority_queue 类中主要提供了 5 个常用方法，它们是：

- void push(const value_type& x)

把元素x插入到优先级队列中。

- void pop()

删除优先级队列的第 1 个值，也就是优先级最高的那个值，前提是队列非空。

- const value_type& top() const

获得优先级队列中的第 1 个元素，返回的是一个引用。

- size_type size() const

返回优先级队列中元素的个数。

- bool empty() const

判断优先级队列是否为空，若为空返回 true，否则返回 false。

下面给出一段简单的示例代码，它说明了上述函数的使用方法，priority_queue 被包含在头文件<queue>中，因此在使用 priority_queue 类之前，需要引入头文件<queue>。

```
#include "stdafx.h"
#include <iostream>
#include <cstdlib>
#include <queue>

using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
    priority_queue<int> Q;
    Q.push(1);
    Q.push(5);
    Q.push(2);
    Q.push(4);
    Q.push(6);
    Q.push(3);

    int size = Q.size();

    for(int i = 0; i< size; i++)
    {
        cout<<Q.top()<<endl;
        Q.pop();
    }

    cout<< endl << Q.empty() <<endl;
    system("PAUSE");
    return 0;
}
```

完成编码后，编译并运行上述程序，运行结果如图 5-9 所示。从中可见优先级队列采用了默认的排序机制对加入队列中的元素进行了排序。默认情况下数值较大的元素具有更高的优先级。事实上，priority_queue 类中包含有很多构造函数，本书略去了这部分内容，有兴趣的读者可以查阅其他资料。但需要说明的是通过不同的构造函数可以构造出具有不同性质的优先级队列，这在实际应用中是非常灵活的。

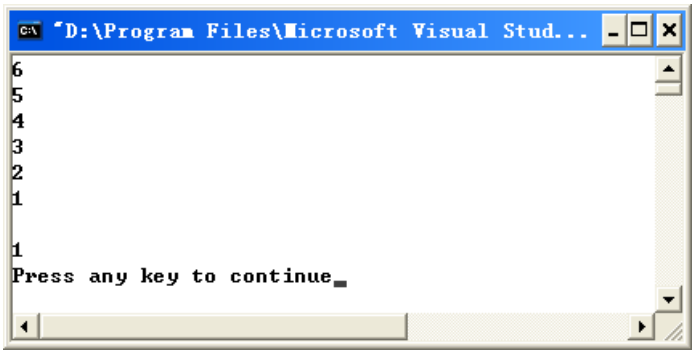


图 5-9　程序运行结果

读者或许会好奇是不是只能使用默认提供的排序规则来限定优先级队列中元素的优先级呢？当然不是。改变优先级定义规则的一种方法是使用自己的函数对象定义。下面这段示例代码在上面例子的基础之上仅做了一些简单的改动，但所构造出来的优先级队列却拥有与前面例子中完全相反的优先级定义方式。

```
#include "stdafx.h"
#include <iostream>
#include <cstdlib>
#include <queue>

using namespace std;

class myCompare{
public:
   bool operator () (const int& a,const int& b)
   {
      return a > b;
   }
};

int _tmain(int argc, _TCHAR* argv[])
{
    priority_queue<int, vector<int>, myCompare> Q;
    Q.push(1);
    Q.push(5);
    Q.push(2);
    Q.push(4);
    Q.push(6);
    Q.push(3);

    int size = Q.size();
    for(int i = 0; i< size; i++)
    {
        cout<<Q.top()<<endl;
        Q.pop();
    }

    cout<< endl << Q.empty() <<endl;
    system("PAUSE");
    return 0;
}
```

完成编码后，编译并运行上述程序，运行结果如图 5-10 所示。

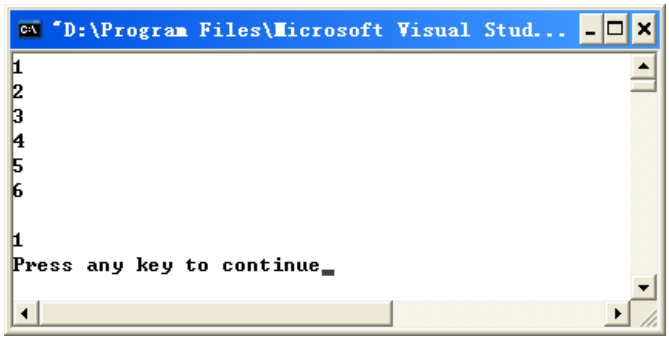


图 5-10　程序运行结果

可见在这个程序中，数值较小的元素具有更高的优先级。通过使用自己的函数对象定义可以定义各种各样的优先级规则。读者不妨亲自动手编写一个程序来模拟基于 LRU 策略的内存页面淘汰过程，如果读者可以充分理解 5.4.2 节和本节所举之程序示例，那么相信实现这个模拟程序应当是非常容易的。

本章参考文献

[1] 左飞，李召恒．C 语言参悟之旅．北京：中国铁道出版社，2010.

[2] 徐甲同，陆丽娜，谷建华．计算机操作系统教程（第 2 版）．西安：西安电子科技大学出版社，2006.

[3] P.J. Plauger, Alexander A. Stepanov, Meng Lee, David R. Musser. 王昕, 译. C++ STL 中文版．北京：中国电力出版社，2002.

第 6 章

递归——老和尚讲故事

递归是强大的问题解决工具，是程序设计中的一种重要思想和机制。递归有助于写出清晰易懂的代码，能有效提高程序的整体风格。本章介绍递归的概念，应用递归的一些原则以及递归与非递归之间的转化等问题，并在此基础上介绍应用递归进行问题求解的两类典型思想, 即分治和回溯。应用这两种思想, 可以方便地解决很多经典问题。

6.1 递归的概念

本节介绍递归的有关概念。我们将从日常生活中典型的包含递归的例子开始描述递归的定义，并分析了应用递归时应当注意的 5 条原则，剖析了递归的工作机制。在介绍递归与非递归的转化时，讨论了应用迭代进行两者间转化的典型例子，这些例子将有助于加深读者对这方面知识的理解。

6.1.1 定义

小提示

理解递归的意义非常重要，它是你在实际问题求解中的一件利器。

什么是递归呢？在数学及程序设计方法学中为递归下的定义是这样的——若一个对象部分地包含它自己，或用它自己来定义自己，则称这个对象是递归的；若一个过程直接或间接地调用自己，则称这个过程为递归的过程。简而言之，递归方法就是直接或间接地调用其自身，递归方法可以用来将一些复杂的问题简化，C++也像其他语言一样支持递归。

在日常生活中，递归的例子是十分普遍的。下面简单举几个例子来阐释递归的概念。

- 字典

字典是递归定义的典型实例。字典中任何一个词都是由“其他词”来解释或定义的，但是“其他词”在被定义或解释的时候又会间接或直接地用到那些由它们定义的词。使用者可以读懂字典中全部词汇的释义，前提是使用者必须掌握少量非常基础的词的含义。当使用者查询某个生僻的词汇时，他发现释义中的某个词汇无法理解，因此他继续在字典中查询那些暂时不能理解的词汇释义。如此继续下去，但此过程最后必然终止，因为最后所有的释义都归结为读者已经了解的常见意思，否则字典使用者将陷入一个无解的问题陷阱中。

- 无穷故事

求职面经点拨

请试回答附录题库中的第 24 题。

大家一定听过那个关于老和尚讲故事的“无穷故事”：从前有座山，山里有个庙，庙里有个老和尚，老和尚正在给小和尚讲故事，故事说“从前有座山，山里有个庙，庙里有个老和尚，老和尚正在给小和尚讲故事，故事说……”。当然这是一个不好的例子，因为它将意味着死循环。递归的能力在于用有限的元素来定义对象的无限集合，所以生活中的递归往往存在这种永无终止的情况。但就程序设计而言，递归是需要有边界条件的。在程序设计语言中应当避免这种无穷调用。

- 科赫曲线

1904 年，瑞典数学家海里格·冯·科赫（Helge von Koch）提出了后来以他名字命名的分形曲线——科赫曲线，这也是最早被描述出来的分形曲线之一。设想一个边长为 1 的等边三角形，取每边中间的三分之一，接上去一个形状完全相似的但边长为其三分之一的等边三角形，结果是一个六角形。再取六角形的每个边做同样的变换，即在中间三分之一接上更小的等边三角形，以此重复，直至无穷，如图 6-1 所示。显然，每次变化后图形的面积和周长都会增加，但是总面积的极限却趋向一个有限值（图形的面积永远不会超过初始三角形的外接圆），而图形的周长却具有无限长度。相比于平常的几何图形，科赫曲线的这种特殊性质显得非常不可思议。此外，科赫曲线还具有极其复杂而精细的自相似结构，即某一个细节放大后将呈现出与整体的惊人相似。科赫曲线的生成方式就是一种递归。

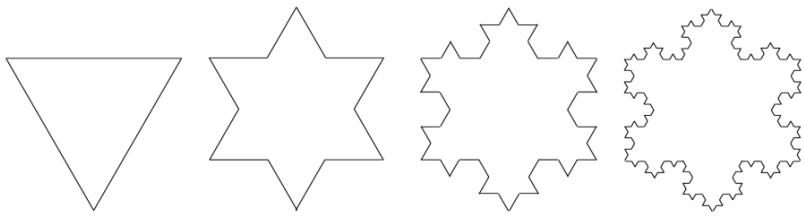


图 6-1　科赫曲线

通常在下面 3 种情况下递归的方法会被用到。

- 定义是递归的。
- 数据结构是递归的。
- 问题的解法是递归的。

首先，来看关于定义是递归的这方面的例子。在数学上常用的阶乘函数、斐波那契序列等，它们的定义和计算都是递归的。例如，阶乘函数的定义为：

$$n! = \begin{cases} 1, & n = 0 \\ n(n-1)!, & n > 0 \end{cases}$$

这个定义十分容易理解而且易于使用，下面来看看 3!是如何根据这个定义求得的：

$$3! = 3 \times 2! = 3 \times (2 \times 1!) = 3 \times [2 \times (1 \times 0!)] = 3 \times [2 \times (1 \times 1)] = 6$$

下面给出了阶乘的递归求解函数。

```
int factorial(int n) {
    if (n == 0) {
        return 1;
    }
    else {
        int value = factorial(n - 1);
        return n * value;
    }
}
```

其次，某些数据结构是递归的，例如，链表就是一种递归的数据结构。从概念上讲，单链表可以递归地定义为一个节点，当该节点的指针域为 NULL 时，就表明此链表是一个单链表，这个节点的指针域也可以指向另一个单链表，而这个单链表具有同样的结构。

此外，作为另外一种常用的数据结构，树也可以采用递归的方式来描述。首先，一棵树要么是空的，要么由根和若干非空子树组成（子树的数目可以为零），且这些子树的根都通过一条边连到根上。每个子树同样具有这样的结构，要么为空，要么由根和若干非空子树组成。特别需要说明的是对于递归的数据结构，采用递归的方法来编写算法是比较简便的。下面举一个编译原理中的例子。

编译程序需要能够对语言句型进行分析。所谓句型分析就是构造某种算法来判断所给的符号串是否为某一文法的句型或句子。对于一个编译程序而言，无论是在词法分析阶段，还是在语法分析阶段，都需要用到句型分析，可见句型分析的重要性。在进行句型分析时，需要通过递归技术构造树结构来解决问题。

所有的程序设计语言都可以写出像$1 * (2 + 3)$这样的算术表达式。下面就以此为例来看看如何描述和判断算术表达式。一个算术表达式可以是如下的任何一种形式。

A. 一个数值常量。

B. 一个表示数值常量或变量的标识符。

C. 一个包含在括号中的算术表达式。

D. 被一个二元运算符分开的两个算术表达式。

像这样的定义在描述程序设计语言的语法时应用十分广泛。任何编译程序的第一步都是使用这些定义来拆解原始编程语言的语句。这个过程的结果被称为分析树。反复应用定义来将一个给定值分解成若干部分，然后再检查每一部分的合法性。基本条件将指导判断某些特定值是否合法。

对于上面的例子首先应用规则D。这样可以把原始输入拆解成 3 个部分。即1、$*$、$(2 + 3)$，并对它们进行分析。结果显示如果以下内容被确定，那么就可以判定原始的输入是否为一个合法的算术表达式了。

(a) 1是一个算术表达式。

(b) $*$是一个二元运算符。

(c) $(2+3)$是一个算术表达式。

通过规则A可以判定(a)是正确的，在已知的二元运算符表中查询*便可判定它是一个二元运算符，(b)是正确的。综上可知为了判定原始输入是一个合法的表达式，仅需判定(c)的合法性即可。而判定(c)需要如下几步。

应用规则C可知，只要判定$2+3$是一个算术表达式就可以判定$(2+3)$是一个算术表达式。为了判定$2+3$是一个算术表达式这里应用规则D将$2+3$分成 3 个部分，即2、+和3。应用上面的规则可知。

(d) 2是一个算术表达式。

(e) +是一个二元运算符。

(f) 3是一个算术表达式。

通过规则A可以很容易判定(d)和(f)的合法性。而在已知的二元运算符表中查询+则可判定它是一个二元运算符。因此所有的条件都表示原始输入是一个合法的算术表达式。整个判定过程不仅认定了原始输入的合法性，而且给出了表达式的结构，如图 6-2 所示为原表达式的分析树。

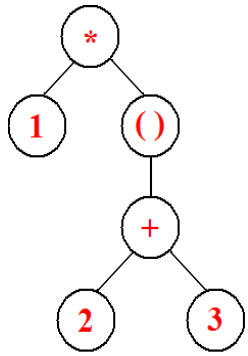


图 6-2　分析树

除了递归的定义和递归的数据结构以外，一些问题的求解方法也可以是递归的，如分治等，关于这一部分的内容将在本章的后续几节中进一步介绍。

6.1.2　应用递归的原则

递归是一个强有力的问题解决工具，但是在定义递归方法时务必谨慎，因为不适当的递归很可能产生一个方法无止尽地调用其自身。下面将介绍一些使用递归的基本原则。

首先是必须有一些“基本条件”（base case）能够采用非递归的方式计算得到，这是使用递归方法的重要前提。基本条件的满足意味着采用递归处理后的子问题可以直接解决时，就停止分解，而这些可以直接求解的问题就叫作递归的“基本条件”。为了使计算最终能够完结，任何递归调用都要朝着“基本条件”的方向进行。考虑前面的前n个整数的递归求和方法。其中“if (n==1)return 1;”就是所谓的基本条件。来看一个复杂

一点的例子，试着找出其中的基本条件。这个例子实现的是二分查找功能。

算法

二分查找算法及其递归实现。

```
#include "stdio.h"
#include <vector>
#include <iostream>

using namespace std;

int binarySearch(const vector<int> & a, const int & x, int low, int high)
{
    if(low>high)
        return -1;

    int mid = (low + high)/2;
    //二分查找法的递归核心部分
    if(a[mid]<x)
        return binarySearch(a, x, mid+1, high);
    else if(x<a[mid])
        return binarySearch(a, x, low, mid-1);
    else
        return mid;
}

int Search(const vector<int> & a, const int & x)
{
    return binarySearch(a, x, 0, a.size()-1);
}

void main()
{
    vector<int> box;

    box.push_back(1);
    box.push_back(4);
    box.push_back(6);
    box.push_back(7);
    box.push_back(8);
    box.push_back(10);
    box.push_back(13);
    box.push_back(21);
    box.push_back(22);
    box.push_back(30);

    const int searchValue = 22;

    int result = Search(box, searchValue);

    if(result == -1)
        cout<<"要查的数字不在数组内！"<<endl;
    else
        cout<<searchValue<<"的位置在第"<<++result<<"位"<<endl;
}
```

之所以举这个例子，那是因为这个例子的基本条件不止一个。而“ if(low>high) ”仅仅

是所谓的基本条件之一。即在局部范围内无法找到待搜索值时就表示搜索失败，递归过程也就结束。此外，当找到待搜索值时，递归过程也会结束。这里的条件不是很明显，因为它采用了一种隐式而非显式的表达方式，即“ else return mid; ”，将这一句改写成“ else if(a[mid]==x) return mid; ”就明显得多了。

在很多情况下，递归的过程可能很长，递归的分支也可能较多，很多初学者往往对此感到困惑。他们总是担心递归的过程是否能够按计划地进行下去。尤其在出现编译错误或逻辑错误时，更是感觉无从下手。但是无论如何在设计递归函数时设法跟踪可能很长的递归调用过程是非常不明智的。于是有学者提出使用递归的另外一条原则，是在设计递归算法时总是假设递归调用是有效的（如果它们最终归结于基本条件）。递归的有效性就像数学归纳法的思想一样。数学归纳法总是在确定初始条件成立的情况下，证明递推关系的成立，进而证明结论的正确性。不妨来看数学归纳法的一个简单例子，证明下列等式。

$$1^2 + 2^2 + \cdots + n^2 = \frac{n(n+1)\,(2n+1)}{6}$$

具体证明过程如下：

（1）当$n = 1$时，等式左边$= 1^2 = 1$，等式右边$= 1 \times (1+1) \times (2+1)/6 = 1$，即左右两边相等，等式成立。

（2）假设当$n = k$时，等式成立，则有当$n = k+1$时，

$$1^2 + 2^2 + \cdots + k^2 + (k+1)^2 = \frac{k(k+1)(2k+1)}{6} + (k+1)^2$$

$$= \frac{(k+1)(2k^2 + 7\mathrm{k} + 6)}{6}$$

$$= \frac{(\,\mathrm{k}+1)(\mathrm{k}+2)(2\mathrm{k}+3)}{6}$$

则证明$n = k+1$时，等式也成立。综上所述，证明原等式成立。

数学归纳法是递归的数学基础。易见，上面例子中证明n等于任意整数时等式成立，也仅仅在证明初始条件成立的基础上论证递推关系的正确性即可，也就是证明当$n = k$时等式成立那么$n = k+1$时等式也成立这个关系。当递推关系得到证明后，不论n等于多少，都无须再关心从1到n之间经过的任何一步。递归的过程也是如此，程序员大可无须关心到每次递归的下一步递归是否能够正确地进行，而仅仅保证基本条件存在，并且所有递归过程的方向都在向着基本条件进行就已经可以了。

接下来要讨论的问题是关于递归的效率的。递归可以使程序的风格明朗清晰，使程序的结构干净利落。但是递归并不是一种经济的做法。先来看一个例子——斐波那契数列的递归实现。

```
int Fib(int n){
    if(n<=1) return n;
    else return Fib(n-1)+Fib(n-2);
```

```
}
```

图 6-3 显示了斐波那契数列的递归调用树。分析可知求解斐波那契数列的调用次数 $\text{NumCall}(k) = 2 * \text{Fib}(k+1) - 1$。这实在是一个无法忽视的开销。事实上，采用非递归的方式求解这个问题的计算量远小于使用递归的方式。一方面递归的过程需要更多的内部调用次数。另一方面，递归总是使用簿记的方式来跟踪前一个调用块的位置，以保证递归后程序能够正确地返回调用点。而记录这些地址也需要消耗一定的时间，因此导致了效率的下降。特别是在递归调用的过程非常长的情况下，对栈空间的开销也很惊人，而栈空间又是十分宝贵的。因此在设计算法时最好做到一定的权衡。

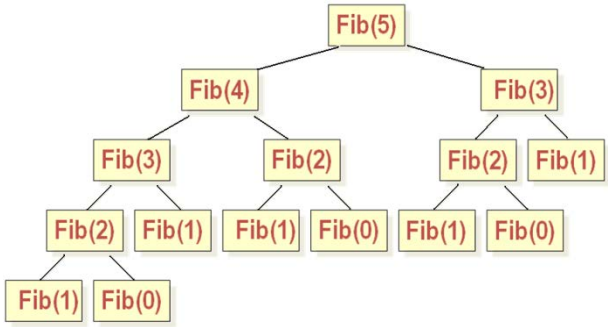


图 6-3　斐波那契数列的递归调用树

最后要注意执行递归调用的顺序问题，首先看下面这个例子。

```
#include "stdio.h"

void recursiveFunction1 ( int num )
{
    if ( num < 5 )
    {
        printf ( "%d \n" , num ) ;
        recursiveFunction1 ( num +1 ) ;
    }
}

void recursiveFunction2 ( int num )
{
    if ( num < 5 )
    {
        recursiveFunction2 ( num +1 ) ;
        printf ( "%d \n" , num ) ;
    }
}

void main()
{
    recursiveFunction1(0);
    recursiveFunction2(0);
}
```

运行结果如图 6-4 所示。

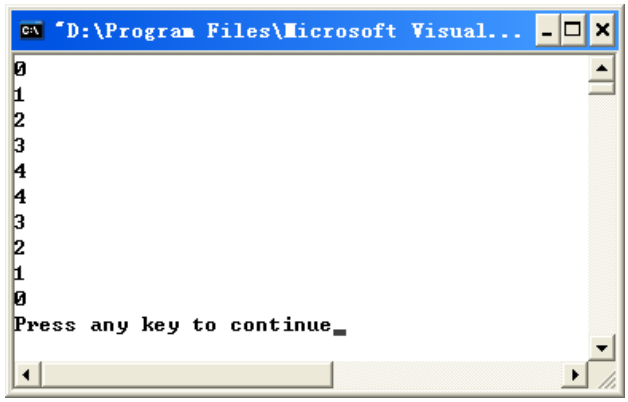


图 6-4　递归函数运行结果

变动调用函数的顺序竟然导致了函数执行顺序的变化，为什么会出现这种情况呢？这就需要对递归过程和递归工作栈有所了解。递归的过程可以总结为以下几点。

- 递归过程在实现时，需要自己调用自己。
- 层层向下递归，退出时的次序正好相反。
- 主程序第一次调用递归过程为外部调用。
- 递归过程每次递归调用自己为内部调用。
- 它们返回调用它的过程的地址不同。

在这个过程中，递归的执行需要一些簿记空间来记录跟踪前一个递归调用，特别对于那些有一长串递归调用的情况，在某种程度上较同等循环而言更加费时，因为簿记工作本身就要消耗一定时间。这个簿记空间就是递归工作栈。也就是说，每一次递归调用时，需要为过程中使用的参数、局部变量等另外分配存储空间。每层递归调用需分配的空间形成递归工作记录，按后进先出的栈组织。这就是工作栈的工作机制。

函数递归时的活动记录可以用图 6-5 表示。

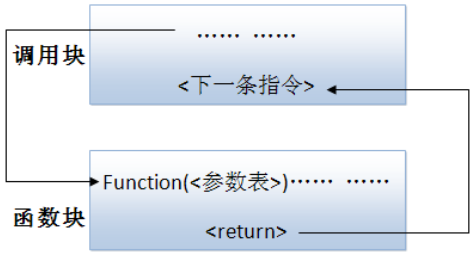


图 6-5　函数递归时的活动记录

根据上面的原理再来分析函数的执行过程。对于函数recursiveFunction1，函数首先执行了一次外部调用recursiveFunction(0)，然后进入函数块执行printf(0)。之后进行第 1 次内部调用recursiveFunction(0 + 1)，进入函数块执行printf(1)。如此继续下去，最

终调用了recursiveFunction(3 + 1)，进入函数块后执行语句printf(4)。当遇到判断条件num < 5时，由于返回了false，不执行任何操作，也就是默认执行了return。返回到前一个调用块的下一条指令，对于函数recursiveFunction1而言，每一个调用块的下一条指令都是return。最终整个过程如图 6-6 所示。

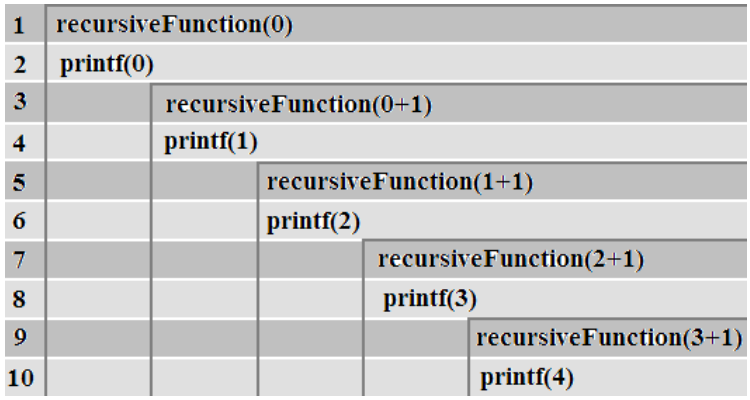


图 6-6　函数 recursiveFunction1 执行过程

对于函数recursiveFunction2，首先执行了一次外部调用recursiveFunction(0)，进入函数块后紧接着又进行了一次内部调用recursiveFunction(0 + 1)，然后依次进行了内部调用直到调用了recursiveFunction(4 + 1)，遇到判断条件num < 5时，由于返回了false，不执行任何操作，也就是默认执行了return。返回到前一个调用块的下一条指令，也就是printf(4)。继续执行下去，于是便不断地返回到前一个调用块的下一条指令，这些指令依次为：printf(3)、printf(2)、printf(1)和printf(0)。最终整个过程如图 6-7 所示。

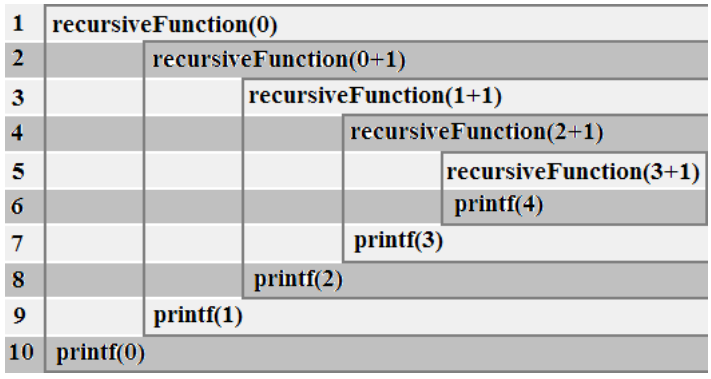


图 6-7　函数 recursiveFunction2 执行过程

下面总结应用递归的 5 条基本原则。

- **基本条件**：递归过程必须一直存在至少一个不使用递归方法解决的条件。
- **进行方向**：任何递归调用都必须向着“ 基本条件 ”的方向进行。
- **正确假设**：总是假设递归调用是有效的。

- **适度原则**：避免使用过多的递归，尤其在效率要求高而递归调用链过长的情况下。
- **顺序问题**：变动递归调用函数的顺序有可能导致整个函数执行顺序的变化。

6.1.3 递归和非递归的转化

递归的特点包括：递归过程简洁、易编、易懂；递归过程效率低，重复计算多；每个递归过程包含两部分，即基本条件和将一个特定的问题削减为一个或多个更简单的问题，直至最终精简为递归终止状态的递归实现部分。

求职面经点拨
请试回答附录题库中的第 25 题。

考虑递归的求解方式带来的低效性，可以尝试将递归的过程改为非递归的过程，本节将介绍这种技术。

在此之前首先说明一些非递归的数学问题也可以转化为递归的描述形式。例如，求前n个整数的和$\mathrm{Sum}(N)$，根据等差数列求和公式可以知道$\mathrm{Sum}(N)=N(N+1)/2$，但是下面这段代码则给出了一个递归的描述方式。

```
int Sum( int n )
{
    if ( n==1 )
        return 1;
    else
        return Sum(n-1)+n;
}
```

需要注意的是并不是所有的非递归问题都可以转化为递归的形式。研究如何将递归问题采用非递归的方式解决更具实际意义。通常单向递归和尾递归可直接用迭代实现其非递归过程，而其他情形必须借助栈实现非递归过程。那么什么是单向递归呢？斐波那契数列的求解就是一个单向递归。单向递归，简单地说就是指递归的过程总是朝着一个方向进行。如图 6-8 所示的递归过程就是一个非单向递归的例子。函数 1 调用了函数 2，而函数 2 又再次调用了函数 1，如此循环往复下去。

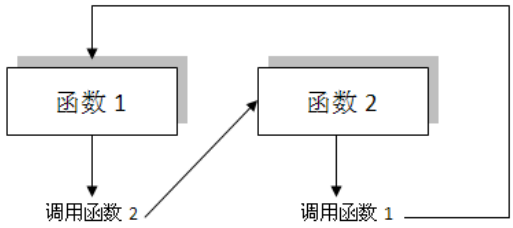


图 6-8 非单向递归举例

单向递归可以用迭代法实现其非递归求解。例如，这里把前面的斐波那契数列的递归求解进行改造就得到了一个迭代法实现的求解函数。

```
int Fib ( int n ) {
    if ( n <= 1 ) return n;
    int twoback = 0;
```

```
    int oneback = 1;
    int Current;
    for ( int i = 2; i <= n; i++ ) {
        Current = twoback + oneback;
        twoback = oneback;
        oneback = Current;
     }
    return Current;
}
```

尾递归函数就是以递归调用作为结尾的函数，它是单向递归的特例，它的递归调用语句只有一个，而且是放在过程的最后。当递归调用返回时，返回到上一层递归调用语句的下一语句，而这个位置正好是程序的结尾，因此递归工作栈中可以不保存返回地址，除了返回值和引用值外，其他参数和局部变量都不再需要，因此可以不用栈，直接采用循环写出非递归过程。图 6-9 是尾递归的示意图。

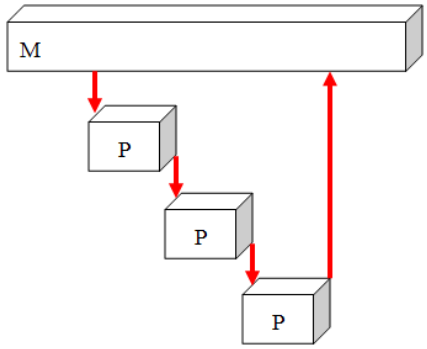


图 6-9　尾递归

与尾递归不同，迭代函数的每个子过程没有发生内部调用，而仅仅是循环并最终返回主程序，如图 6-10 所示。

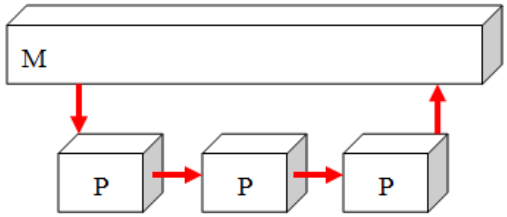


图 6-10　迭代

考虑前面用作例子的阶乘函数，它不就是一个尾递归，因为在它收到递归调用的结果之后，必须在返回调用之前再做一次乘法运算。但是阶乘函数可以转化成一个尾递归函数，如下面这种改造方式就把原来的函数转化成一种尾递归的形式了。

```
int factorial(int acc , int x)
{
    if (x <= 1)
        return acc;
    else
```

```
        return factorial ( x * acc, x - 1 );
}
```

通常认为递归较迭代而言具有更好的可读性，然而某些情况下似乎并非如此。上面采用尾递归改写的阶乘函数就是一个很好的例子，它并不具有很好的可读性反而让人不太容易理解。因此，这里也仅仅是提出这样一种转换的思路，它的实际应用意义不大。在实际的编程中，考虑更多的是效率与可读性的平衡。递归的效率通常要低于迭代，但必须要在保证程序可读性更好的情况下使用递归，否则就得不偿失。

算法

用欧几里得算法求解最大公约数。请注意其递归和非递归实现之间的转化。

下面的求解最大公约数的欧几里德算法函数是尾递归的。

```
int gcd(int a, int b)
{
    if (b == 0)
        return a;
    else
        return gcd(b, a%b);
}
```

而下面这个求解函数则将尾递归用迭代法来替换。

```
int gcd ( int a, int b )
{
    while ( b != 0 )
    {
        int r = a % b;
        a = b;
        b = r;
    } return a;
}
```

尾递归的重要性在于进行尾递归调用时，调用者的返回位置不需要被存于调用栈里，当递归调用返回时，它直接分支到先前已保存的返回地址。因此，在支持尾递归优化的编译器中，尾递归在时间和空间上都比较划算。

迭代算法通常需要一个临时变量，这无疑导致了程序的可读性降低，即使有欧几里德算法的某些必要知识，试图通过简单的检查来理解其迭代求解过程也是比较困难的。

但是，在阶乘例子中迭代实现好像比递归要快些。这对于求解最大公约数的欧几里德算法的执行是很明显的。因为迭代函数不像递归函数那样需要考虑函数调用的支出，尤其当这种调用的数目巨大时，消耗也将是非常惊人的。另一个选择迭代而非递归的可能原因是，在当今的程序设计语言中，对于一个线程来说可用的栈空间通常比可用的堆空间要少得多，而递归算法则相对迭代算法需要更多的栈空间。

6.2 分治法

分治的思想古已有之。秦灭六国，统一天下正是采取各个击破、分而治之的原则。而分治思想在算法设计中也是非常常见的。当一个问题规模较大不易求解的时候，就可以考虑将其分成几个小的模块，逐一解决。因为采用分治处理的问题，其各个小模块

通常具有与大问题相同的结构，这种特性也就使得递归技术有了用武之地。

6.2.1　分治法简述

小提示
算法设计思想之分治法。

分治法（Divide and Conquer）的基本思想是将一个规模较大的问题分解为若干个规模较小的子问题，同时要求这些子问题相互独立且与原问题相同。分治是算法设计中的一个重要思想。在解决具体问题时，反复使用分治手段，可以使原问题的规模不断缩小，直到子问题小到很容易解出，那么原来的问题也就迎刃而解了。由于分治法产生的子问题往往是原问题的较小模式，这就为使用递归技术提供了方便。

在应用分治思想设计具体算法时，需要考虑到底应该把原问题分解为多少个子问题。从大量实践和经验角度来说，通常子问题的规模应当大致相同，也就是说把一个问题分成大小相等的若干个子问题的处理方式较为有效。在更多的情况下，问题总是被分为两个子问题，以期达到一种相对的平衡。

分而治之的思想最简单的情况就是二分法。假想总是把问题一分为二地变成两个更小规模的子问题，并按照这样的程序一直分解下去直到问题小到能够轻而易举地解决，那么大问题也就随之而解了。二分法的例子是非常之多的，前面的二分查找就是一个很好的例子。在电影《永不消逝的电波》中敌人也应用了这种方法。敌人发现有地下党的信息台在向外发送情报，但是敌人不能确定其准确位置。于是敌人尝试将城区分成两部分，比如东城区和西城区。先将东城区停电，再看是否还有电台在工作，如果有则证明电台位于西城，否则就位于东城。再继续将东城或西城进而分成两个部分，如此继续下去最后就可以准确地发现电台的位置了。

下面来看一个“伪币问题”。假设有一个装有 16 个硬币的袋子。16 个硬币中有且只有一个是伪造的，并且那个伪币要较真的硬币轻些。现在考虑如何找出这个伪造的硬币。这里仅提供一台可用来比较两组硬币重量的天平。通过这台仪器，可以知道两组硬币的重量是否相同。

可供选择的第 1 种方法描述如下：首先随意挑选两枚硬币，标记为 1 和 2。比较硬币 1 与硬币 2 的重量。假如硬币 1 比硬币 2 轻，则硬币 1 是伪造的；否则如果硬币 2 更轻，那么硬币 2 是伪造的。假如两枚硬币重量相等，则说明两枚硬币都是真的，则继续比较硬币 3 和硬币 4。同样，假如有一个硬币轻一些，那么寻找伪币的任务就完成了。否则继续比较剩余硬币。按照这种方式，最多可以通过 8 次比较来判断伪币的存在并找出这一伪币。

另外一种方法就是利用分而治之的方法。假如把 16 枚硬币的例子看成一个大问题。首先，把这一问题分成两个小问题。随机选择 8 枚硬币作为第一组，标记为 A 组，剩下的 8 枚硬币作为第二组，标记为 B 组。然后，用天平来比较 A 组硬币和 B 组硬币的重量。假如两组硬币重量不相等，则伪币存在于较轻的那一组硬币中。接着重复这一步骤，将较轻的一组再次分成两组，用天平比较后取其中较轻的一组。如此下去最多通过 4 次比较就能找出这一伪币。而如果仅需判断是否存在伪币，那么事实上仅需一

次比较就足够了。

再来回顾一下前面曾经举过的例子——求斐波那契序列的函数Fib(n)的递归定义。计算斐波那契序列函数对应的递归过程定义如下。

```
int Fib(int n){
    if(n<=1) return n;
    else return Fib(n-1)+Fib(n-2);
}
```

可见每次求一个斐波那契数，都将其分为两个部分，一个子问题是求解前一个斐波那契数，另一个子问题是求解再前一个斐波那契数。总之，分治法使用两个递归的函数调用。这两个递归函数调用不断地将问题空间拆解成两个部分。当递归过程到达基本条件时，子问题就可以直接解决了。这些子问题的解决结果再组合起来最终推出原始问题的答案。下面将考察若干采用分治法解决的经典问题。

经典问题求解

汉诺塔问题。

6.2.2 汉诺塔问题

汉诺（Hanoi）塔问题最初源于法国数学家爱德华·卢卡斯（Édouard Lucas）于 1883 年发明的一个休闲数学游戏。但也同时流传着一个与之相关的古老传说。据说在印度的一座神庙里，一块黄铜板上立着三根金刚石柱子。印度教的主神梵天创造世界的时候，在其中一根柱子上从下到上地穿好了由大到小的 64 片黄金圆盘，这就是所谓的汉诺塔。不论白天黑夜，总有一个僧侣在按照下面的法则移动这些圆盘：一次只移动一片，而且不管在哪根柱子上，小圆盘必须置于大圆盘上面。僧侣们预言，当所有的圆盘都从梵天穿好的那根柱子上移到另外一根柱子上时，世界就将毁灭。后人已经无法确定卢卡斯是自创的这个游戏，还是他从别人那里听闻过这个传说。

汉诺塔是一个需要用递归方法进行求解的典型问题。我们考虑简化的情况，即只有 3 个盘子的汉诺塔模型，如图 6-11 所示。

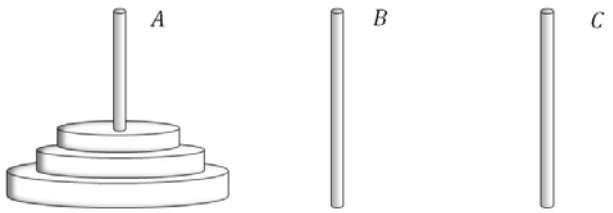


图 6-11 汉诺塔模型

下面以三阶汉诺塔的移动为例来看看这个问题应该如何求解。如图 6-12 所示为三阶汉诺塔的求解过程。

如果想移动底部的大号盘子，则必须移开它上面的中号盘子，而若想移动中号盘子，又必须移动上面的小号盘子。所以首先将小号盘子从A移到C，再将中号盘子从A移到B。然后将小号盘子从C移到B，并将大号盘子从A移到C，然后再考虑将中号盘子从B移到C。所以需要将小号盘子从B移到A，然后再将中号盘子从B移到C，最后再将小号盘子从A

移到C，则整个过程就完成了。当盘子数增加时，只要按着这样的递归规则来移动，最初的大问题就会逐个分解为规模更小的问题，求解难度也随之降低。

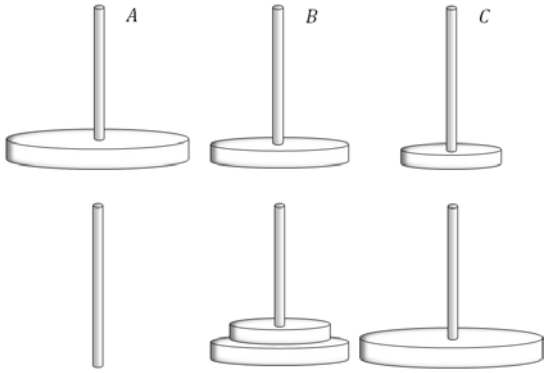


图 6-12　三阶汉诺塔的求解过程

推广开来，当盘子数目为 1 时，只要将盘子从A直接移动到B上即可。当$n > 1$时，则需要利用C来作为辅助的腾挪空间。这时需要想办法将$n-1$个较小的圆盘依照规则从A移到C，再将剩下的最大的盘子从A移到B。最后再将$n-1$个小盘依照规则从C移到B。如此下去，n个圆盘的移动问题就可以分解为两次$n-1$个圆盘的移动问题，这也就体现出了分而治之的策略。下面给出了求解汉诺塔问题的示例程序。

```
#include "stdafx.h"
#include <fstream>
#include <iostream>

using namespace std;

//盘子的数目
#define numOfDisks 10

//在文本文件 out.txt 中输出结果
ofstream fout("out.txt");

void Move(int n,char x,char y)
{
    fout<<"move "<<n<<" from "<<x<<" to "<<y<<endl;
}

//递归求解
void Hannoi(int n,char a,char b,char c)
{
    if(n==1)
        Move(1,a,c);
    else
    {
        Hannoi(n-1,a,c,b);
        Move(n,a,c);
        Hannoi(n-1,b,a,c);
    }
```

```
}

int _tmain(int argc, _TCHAR* argv[])
{
    fout<<"The solution for Hanoi, when the number of disks is "<<
        numOfDisks<<endl;
    Hannoi(numOfDisks,'a','b','c');
    fout.close();
    cout<<"The End! Please Check out.txt."<<endl;
    system("pause");
    return 0;
}
```

经典问题求解

传染病问题。

6.2.3 传染病问题

根据世界卫生组织的报告，传染性疾病是导致人类死亡的主要原因之一。仅在 1998 年一年中，就有超过一千七百万人死于传染病以及寄生虫病。其中例如急性呼吸道传染病、肺结核、艾滋病和疟疾都是严重威胁人类健康的恶性传染病。该报告还预测传染病将继续会导致成千上万人的死亡，尤其在发展中国家这种情况将尤为明显。

世界范围内的医疗专家和科学团体一直在不同地区采用各种新方法对抗传染病，由此一些新的领域也随之诞生。传染病流行学作为一门新兴学科，致力于研究对于传染病发展传播起到相当作用的各种因素之间的相互关系。它的研究范围很广，与免疫学、分子生物学、遗传学和计算科学都互有交叉。该领域的一个研究焦点是研究在单一有机体内传染病发展的影响因子模型，进而对整个人群的传染病发展的影响因子模型进行研究。

现在需要完成一个用以评估某组织样本中感染水平的程序。输入的数据用来表示一个矩形的组织样本。这个矩形的组织样本用一个由 0 和 1 组成的二维阵列表示。组织中的某一部分可能被感染也可能没有被感染。正如同图 6-13 中所展示的那样，当组织中的一个细胞被感染后，那就被 1 标识。相反，一个健康的细胞则用 0 来标识。易见，图 6-13 中存在 3 个菌群。最小的那个位于左下角，仅由两个细胞组成。另外一个稍大一点的菌群则位于图像的右侧，由 3 个细胞组成。最大的菌群则由 8 个被感染的细胞组成。为了估算机体受感染的程度，该程序需要在接收一个坐标输入后，以该点为中心向周围的 8 个方向进行搜索，并检查周围的细胞是否被感染。假设当输入为(1,1)时，则把得到的菌群用+1表示。

这同样是一个运用分治法解决的典型问题。它与汉诺塔问题的区别在于它的子问题数不再是 2 个，而是 8 个，因为每个点都有 8 个临近点（除了边缘点以外）。另外，这个问题在数字图像处理中也会遇到。它应该可以看作是种子算法的一种变形，因此也建议有兴趣的读者自己尝试着求解这个问题看看。

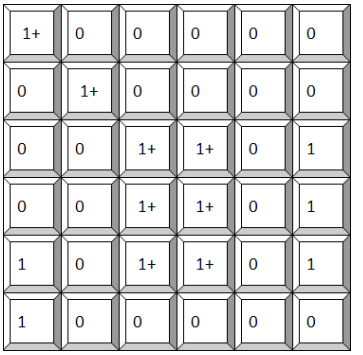


图 6-13　被感染的组织

需要说明的是，这个问题的关键在于运用递归搜索时，一定要注意排除那些已经检查过的点，否则程序将陷入一个死循环之中，另外同样需要注意边界的检查避免越界操作。下面给出执行测试用例的主程序文件代码。其余函数的代码清单可在本书的在线支持网站中获得。

```
#include <iostream>
#include <string>
#include <cstdlib>

#include "grid.h"

using namespace std;

#define ROWS 6
#define COLS 6
#define TOTAL 13 // 被感染点的总数

// 标记被感染点的坐标的数组
int theCity[TOTAL][2] = {
    {0,0},
    {1,1},
    {2,2},
    {2,3},
    {2,5},
    {3,2},
    {3,3},
    {3,5},
    {4,0},
    {4,2},
    {4,3},
    {4,5},
    {5,0}
};

int main () {

    grid *g;
    int col;
    int row;
```

```
    g = new grid ((int*)theCity, ROWS, COLS, TOTAL);

    cout << "请输入要检测点的坐标格式: x,y" << endl;
    scanf("%d,%d", &row, &col);

    cout << "The colony including the cell at "
         << "(" << row << "," << col << ")"
         << " has an area of " << g->count (row,col) << " units." << endl;

    cout << *g << endl;

    delete g;

    return EXIT_SUCCESS;
}
```

6.3 回溯法

回溯法也叫试探法，它是一种系统地搜索问题之解的方法。“回溯”，这个词本身有逆流而上的意思。而回溯的过程正是当某一种可能的试探结果否定了该可能路径的正确性后返回先前的某个状态继续进行其他可能性的试探的过程。可以说回溯策略并非按照某种固定的计算方法来设计算法，而是通过尝试和纠正错误来寻找答案。本节将使用回溯思想解决若干经典问题并通过它们来说明使用回溯的基本思路。

小提示

算法设计思想之回溯法。

6.3.1 回溯法简述

回溯是一种比较简单、比较常用的搜索策略。它的基本思想是假设某问题的解决步骤可能有N步，且每一步的解决方法又可能有M种，那么就按照某种顺序依次试探每一步中的各种方法，一旦某一步的所有方法都失效，那么就返回上一步继续试探上一步骤的其他$M-1$种方法。简而言之就是从一条路往前走，能进则进，不能进则退回来，换一条路再试。

通常用回溯法解决问题的一般步骤为：首先，定义一个解空间，它包含问题的解，也就是每一步所采用的各种方法。然后利用适于搜索的方法组织解空间，再利用深度优先法搜索解空间。并利用限界函数避免移动到不可能产生解的子空间。这里问题的解空间通常是在搜索问题的解的过程中动态产生的，这是回溯法的一个重要特性。而其中全面访问所有可能的情况又可以分为两种：一是不考虑给定问题的特有性质，按事先定好的顺序，依次运用规则，即盲目搜索的方法；另一种则考虑问题给定的特有性质，选用合适的规则，提高搜索的效率，即启发式的搜索。

经典问题求解

迷宫问题

6.3.2 迷宫问题

迷宫问题是应用回溯法解决的典型问题。迷宫最早出现在古希腊神话中。传说在远古时代，麦诺安帝国国王弥诺斯在克里特岛建造了一座有无数宫殿的迷宫，迷宫中道路

曲折纵横，谁进去都别想出来，而且在迷宫的纵深处还有一个牛头怪。弥诺斯要求雅典每隔 9 年送 7 对童男童女到克里特岛。这些童男童女就是供奉给牛头怪吃的。

来自雅典的王子忒修斯立志要为民除害，于是他与当年选献的童男童女一起来到克里特岛，忒修斯上岸后爱上了美丽的阿里阿德涅公主。当公主知道忒修斯的使命后，就送给他一把魔剑和一个线球，以免忒修斯受到牛头怪的伤害。

勇敢的忒修斯一进入迷宫，就将线球的一端拴在迷宫的入口处，然后放开线团，沿着曲折复杂的通道，向迷宫深处走去。最后，他终于找到了牛头怪，并用阿里阿德涅公主给的剑，奋力杀死了牛头怪。然后，他带着童男童女，顺着线路走出了迷宫。在阿里阿德涅公主的帮助下，忒修斯等人一起逃出了克里特岛。

现假设迷宫由一系列交叉路口组成，从一个入口进入后可以沿着 3 个方向走：向左、向前或者向右。迷宫的路径由一系列标识路口的序号组成。为了求解迷宫问题，需要用到回溯的方法，当沿着某一路径一步步走向出口却发现进入死胡同之时，就回溯一步或多步，寻找其他可走的路径。

如图 6-14 所示为一个迷宫。从入口 1 的位置出发，沿东走到节点 2 时发现有两条路。标记节点 2，然后继续向东走，直到走到尽头无路可走时回溯到节点 2 尝试另外一种可能的走法，即向右转，沿南走到节点 4，这时有 3 个选择，可以向右、向前或者向左。结果发现向左或向右都走不通，则再次退回到节点 4 向前，沿南走到节点 5。如此继续下去，则可以最终到达出口 10 的位置。下面给出了求解迷宫问题的示例程序。

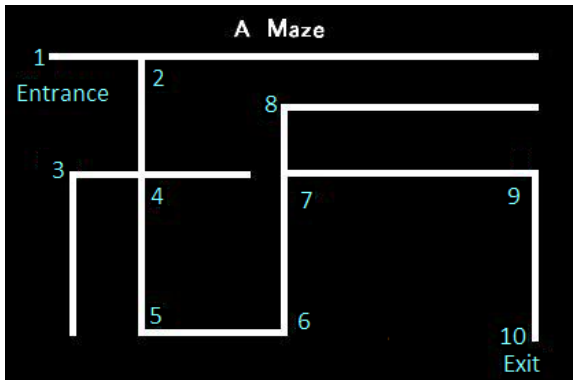


图 6-14　迷宫

```
#include <vector>
#include <iostream>
using namespace std;

// 把迷宫表示为 n 个有编码路口的集合
// 定义路口类
class Crossing
{
public:
    // 0 为不通路口的 3 个方向
    int turn1;
```

```
    int turn2;
    int turn3;

public:
    Crossing(int turn1, int turn2, int turn3)
    {
        Crossing::turn1 = turn1;
        Crossing::turn2 = turn2;
        Crossing::turn3 = turn3;
    }
};

// 定义迷宫类
class Maze
{
private:
    int exit; //出口编码
    vector<Crossing> crossings; //路口集合
    vector<int> result;

public:
    Maze(int the_exit, vector<Crossing> the_crossings)
    {
        exit = the_exit;
        crossings = the_crossings;
    }
    findExit(int entrance); //迷宫求解
    getResult(); //取得迷宫解并打印

};

//迷宫求解核心算法
Maze::findExit(int entrance)
{
    if(entrance > 0)
    {
        if(entrance == Maze::exit)
        {
            result.push_back(entrance);
            return 1;
        }
        if(findExit(crossings[entrance].turn1))
        {
            result.push_back(entrance);
            return 1;
        }
        if(findExit(crossings[entrance].turn2))
        {
            result.push_back(entrance);
            return 1;
        }
        if(findExit(crossings[entrance].turn3))
        {
```

```
            result.push_back(entrance);
            return 1;
        }
    }

    return 0;
}

Maze::getResult()
{
    findExit(1);

    for(int i = result.size(); i>0; i--)
        cout << result[i-1] << "->";
    cout << "Exit" << endl;
}

void main()
{
    // 创建一个迷宫 9 个路口，出口为 10
    Crossing c1(2,0,0);
    Crossing c2(4,0,0);
    Crossing c3(0,0,0);
    Crossing c4(3,5,0);
    Crossing c5(6,0,0);
    Crossing c6(7,0,0);
    Crossing c7(8,9,0);
    Crossing c8(0,0,0);
    Crossing c9(10,0,0);
    Crossing c0(0,0,0);

    vector<Crossing> crossings;
    crossings.push_back(c0);
    crossings.push_back(c1);
    crossings.push_back(c2);
    crossings.push_back(c3);
    crossings.push_back(c4);
    crossings.push_back(c5);
    crossings.push_back(c6);
    crossings.push_back(c7);
    crossings.push_back(c8);
    crossings.push_back(c9);

    Maze newMaze(10, crossings);
    newMaze.getResult();

    return;
}
```

由此例可以简单总结出应用回溯法的一般思路。即首先必须明确定义问题的解空间。而问题的解空间应当至少包含该问题的一个解。对于迷宫问题，各路口的集合就是问题的解空间。对于每个节点遍历尝试的可能性最多不超过 3 种，即左转、右转和向前，仅当这 3 种可能性都被否定时才回溯至前一状态。在定义了问题的解空间之后还应当

考虑如何将解空间进行有效的组织，以使得回溯法能够方便地搜索这些子空间中的节点。在必要的时候还应当注意优化搜索的策略以提高算法的实时性。

当以上准备完成之后就从出发点开始，以深度优先的方式对整个解空间进行搜索。该出发点随即被更新为当前的扩展节点。也就是从一个可能的路径进行深入以产生下一个新的节点，并将新的节点更新为扩展节点。一旦当前的扩展节点既不能得出整个问题的一个解，也不能再继续向更深的方向进行搜索，那么就返回上一个节点并将上一个节点重新更新为新的扩展节点，再尝试另一种可能性。回溯法正是采用这种工作方式以递归为基础在解空间内开展系统的搜索工作，直到求出问题的解或者表明问题无解为止。需要说明的是因为回溯法是对解空间的深度优先搜索，所以可以考虑使用树结构的递归遍历方式完成搜索工作。当然这并非是唯一的途径，也可以考虑使用树结构的非递归遍历方法，那样整个回溯过程将以迭代的形式完成。

经典问题求解

八皇后问题。

6.3.3 八皇后问题

八皇后问题是一个古老而著名的问题，它是回溯法的典型例题。该问题最早是由德国棋手马克斯·贝瑟尔（Max Bezzel）于 1848 年提出。之后陆续有数学家对其进行研究，其中包括德国数学家卡尔·弗里德里希·高斯（Karl Friedrich Gauss）和格奥尔格·康托（Georg Cantor），该文是这样描述的：在 8 行 8 列的国际象棋棋盘上摆放着 8 个皇后。若 2 个皇后位于同一行、同一列或同一对角线上，则称它们为互相攻击。在国际象棋中皇后是最强大的棋子，因为它的攻击范围最大，图 6-15 显示了一个皇后的攻击范围。

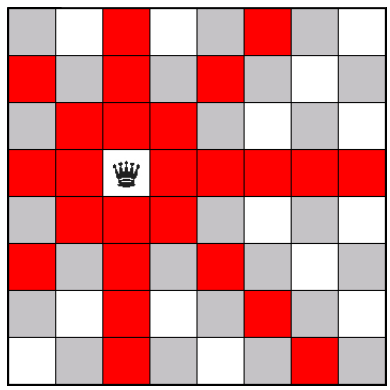


图 6-15 皇后的攻击范围

现在要求使这 8 个皇后不能相互攻击，即任意 2 个皇后都不能处于同一行、同一列或同一对角线上，问有多少种摆法。高斯认为有 76 种方案。1854 年在柏林的象棋杂志上不同的作者发表了 40 种不同的解，后来有人用图论的方法解出 92 种结果。现代教学中，把八皇后问题当成一个经典递归算法例题。图 6-16 显示了两种 8 个皇后不相互攻击的情况。

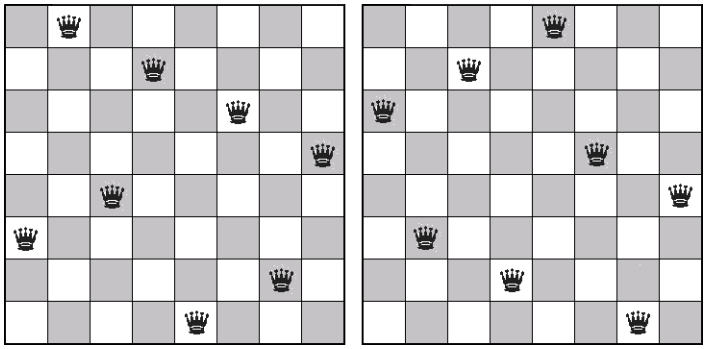


图 6-16　8 个皇后不相互攻击的情况

现在来看如何使用回溯法解决八皇后问题。这个算法将在棋盘上一列一列地摆放皇后直到 8 个皇后在不相互攻击的情况下都被摆放在棋盘上，算法便终止。当一个新加入的皇后因为与已经存在的皇后之间相互攻击而不能被摆在棋盘上时，算法便发生回溯。一旦发生这种情况，就试图把最后放在棋盘上的皇后移动到其他地方。这样做是为了让新加入的皇后能够在不与其他皇后相互攻击的情况下被摆放在棋盘的适当位置上。

如图 6-17 所示的情况，尽管第 7 个皇后不会与已经放在棋盘上的任何一个皇后发生攻击，但仍然需要将它移除并发生回溯，因为无法为第 8 个皇后在棋盘上找到合适的位置。

算法的回溯部分将尝试移动第 7 个皇后到第 7 列的另外一点来为第 8 个皇后在第 8 列寻找一个合适的位置。如果第 7 个皇后由于在第 7 列找不到合适的位置而无法被移动，那么算法就必须去掉它并回溯到第 6 列的皇后。最终算法不断重复着摆放皇后和回溯的过程直到找到问题的解为止。

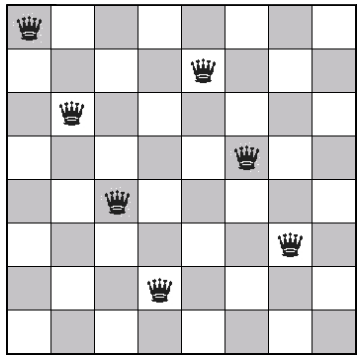


图 6-17　需要发生回溯的情况

由于回溯法也是在试图搜索整个解空间中的所有可能的选择，于是有人会误认为回溯法与穷举法差不多，但事实上回溯法要较穷举法在效率上高出很多。这里就以一种简单的估算方法来对八皇后问题进行一下分析。首先采用穷举法，那么可以容易得出该问题解空间的节点总数为 109601。然后在使用回溯法的前提下随机选取 20 条路径，分别估计它们的节点个数并求得总数的平均值。因为已知八皇后问题共有 92 种解，所

以选择 20 种随机路径进行估计所得出的结果已经可以较为接近实际数值。经过计算得出回溯法产生的节点数的平均值约为 1740，这相对于 109601 不足 2%。可见回溯法作为一种跳跃性和系统性相结合的搜索方法是具有较高效率的。

下面给出了求解八皇后问题的示例程序。

```
#include <conio.h>
#include <iostream>

using namespace std;
// 首先 要求皇后不冲突，那么每行只应该有一个皇后
// 用 queens[]数组存储每个皇后的位置
// 例如: queens[m] = n 表示第 m 行的皇后放在第 n 列上

#define MAX 8

int sum = 0;
class QueenPuzzle
{
    int queens[MAX]; // 存储每行皇后的列标

public:
    void printOut(); // 打印结果
    int IsValid(int n); //判断第 n 个皇后放上去之后，是否合法
    void placeQueen(int i); // 递归算法，放置皇后
};

void QueenPuzzle::printOut()
{
    for(int i=0; i<MAX; i++)
    {
        for(int j=0; j<MAX; j++)
        {
            if(j == queens[i])
                cout << "Q ";
            else
                cout << "0 ";
        }
        cout << endl;
    }
    cout << endl << "按 q 键盘退出，按其他键继续" << endl << endl;

    if(getch() == 'q')
        exit(0);
}

// 在第 i 行放置皇后
void QueenPuzzle::placeQueen(int i)
{
    for(int j=0; j<MAX; j++)
    {
        // 如果全部放完了，输出结果
```

```
        if(i == MAX)
        {
            sum ++;
            cout << "第" << sum << "组解:" << endl;
            printOut();

            return;
        }

        // 放置皇后
        queens[i] = j;

        // 此位置不能放皇后，继续试验下一位置
        if(IsValid(i))
            placeQueen(i+1);
    }
}

//判断第 n 个皇后放上去之后，是否合法，即是否无冲突
int QueenPuzzle::IsValid(int n)
{
    //将第 n 个皇后的位置依次于前面 n－1 个皇后的位置比较
    for(int i = 0 ; i < n ; i++)
    {
        //两个皇后在同一列上，返回 0
        if(queens[i] == queens[n])
            return 0;

        //两个皇后在同一对角线上，返回 0
        if(abs(queens[i] - queens[n]) == (n - i))
            return 0;
    }

    //没有冲突，返回 1。
    return 1;
}

void main()
{
    QueenPuzzle queen;
    queen.placeQueen(0);
    cout << "共" << sum << "组解" << endl;
}
```

本章参考文献

[1]　王晓东．计算机算法设计与分析（第 3 版）．北京：电子工业出版社，2004.

[2]　蒋立源，康慕宁．编译原理（第 3 版）．西安：西北工业大学出版社，2005.

[3]　张乃孝．算法与数据结构——C 语言描述（第 2 版）．北京：高等教育出版社，2006.

第 7 章

树——从红楼梦说起

历史上最早利用树这一概念去解决问题的人是德国物理学家古斯塔夫·罗伯特·基尔霍夫（Gustav Robert Kirchhoff）。1847 年，基尔霍夫在研究电气网络时，首先提出了“树”、“支撑树”和“连通图”等基本概念，此外，基尔霍夫还提出了计算任意图中生成树个数的基本定理——基尔霍夫矩阵树定理。树结构应用于计算机领域由来已久。早期计算机的机器语言通常采用三地址码来表示算术表达式的计算，而三地址码就等价于一个二叉树。此后，各种树结构同大量的计算机应用紧密地联系在一起。由于树这种结构能够很好地体现出一种非线性化的分层结构，因此在实际中具有非常广泛的应用。

7.1 认识树这种结构

在现实世界层次化的数据模型中，数据与数据之间的关系纷繁复杂。其中很多关系无法使用简单的线性结构表示清楚，比如祖先与后代的关系、整体与部分的关系等。于是人们借鉴自然界中树的形象创造了一种强大的非线性结构——树。树是计算机科学中使用最为广泛的数据结构之一，本节介绍与树有关的一些基本概念。

7.1.1 基本定义

问题导入

“贾不假，白玉为堂金作马”。面对复杂的贾家人物关系，若是不用树，如何才能讲清楚？

在给出树的定义之前，先来看一个生活中的例子：中国古典文学名著《红楼梦》以男女主人公的爱情悲剧为主线，描写了贾、王、史、薛四大封建家族的荣衰，展示了广阔的社会生活视野，囊括了多姿多彩的世俗人情，将我国古典文学的艺术成就推向了最高峰。书中人物众多，社会关系复杂，如果使用简单的线性结构来存储这些人物，对于他们之间关系的表示是非常不利的。如果能够使用一种结构，既将所有人物都存储在内，又能够通过层次化的表示方法来体现人物之间的社会关系，那么一个看似错综复杂的问题就变得简单明了了。

图 7-1 采用树这种存储结构来表示红楼梦中贾家的族谱（不含配偶），从中可以清晰地看出贾家的人物关系。在这个模型中，树结构的优势被充分地体现了出来。

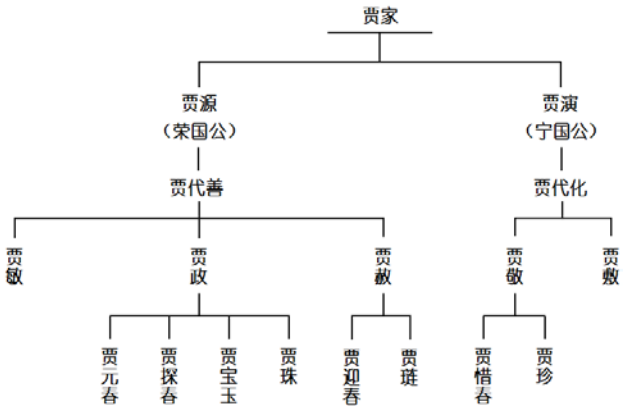


图 7-1　贾家人物关系

从上面这个例子中，读者应当已经隐约地有了一个树结构的形式化描述定义。这种描述告诉读者树应当是一种体现分层系统的非线性结构，它应当由一些点组成，且这些点总是体现出一对多的映射关系。在此基础之上，我们给出树的定义如下，该定义是一种基于递归的表述。

树是由n个节点组成的有限集合T，其中$n \geqslant 0$。特别地，若$n = 0$，称为空树，如果$n > 0$则T应当满足的条件是：

（1）有一个特定的称为根（root）的节点，它只有直接后继，没有直接前驱。

（2）除根以外的其他节点被划分为m个互不相交的有限集合$T_1, T_2, \cdots, T_m$，其中$m \geqslant 0$每个集合又是一棵树，并且称为根的子树。每棵子树的根节点有且仅有一个直接前驱，但可以有 0 个或多个直接后继。

人们为了有效表示现实世界中某些非线性层次关系，而从自然界树的形象中抽象出了这种非线性数据结构。树形结构的逻辑表示形如一棵倒立生长的树木，如图 7-2 所示。可见该树是一个由 9 个节点组成的树，其中A是该树的根节点，它位于树的顶部。除了A之外其余的各个节点被分成了两个互不相交的子集：$T_1 = \{B, D, E, F\}$，$T_2 = \{C, G, H, I\}$。它们即为A的子树。树中亦包含有递归的结构，例如在子树T_1中，B又是其根节点，而且除了根节点B之外的其他节点又被分成了 3 个互不相交的子集：$T_{11} = \{D\}$，$T_{12} = \{E\}$，$T_{13} = \{F\}$。它们 3 个都是只有一个根节点的树，没有子树。

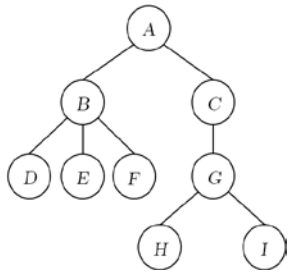


图 7-2　树的示意图

7.1.2 一些术语

为便于后续讲述，下面给出一些与树有关的常用术语及定义。

节点：节点是树的基本构成单位，它由数据项及指向其他节点的分支共同组成。例如在图 7-2 中，共有 9 个节点。

度：树上任一节点所拥有的子节点的数目称为该节点的度，或度数。例如在图 7-2 中，节点 A 的度数为 2。特别地，一棵树中所有节点的度的最大值称为该树的度。

叶子节点：也称作终端节点，即度为 0 的节点。例如在图 7-2 中，节点H和I即为叶子节点。

分支节点：也称作非终端节点，度大于 0 的节点，也就是除了叶子节点以外的其他节点。

父节点：如果树中节点A是节点B的直接前趋，则称A是B的父节点。例如在图 7-2 中，节点A是节点B和C的父节点。

子节点：如果节点A是节点B的父节点，那么节点B就是节点A的子节点。也就是说，如果树中节点B是节点A的直接后继，那么就称B是A的子节点。

兄弟节点：父节点相同的节点互称为兄弟节点。例如在图 7-2 中，D、E、F互为兄弟节点，但需要注意的是F和G不是互为兄弟节点的关系。

子孙节点：树中某一节点的所有子节点，以及这些子节点的子节点都是该节点的子孙节点。从递归的角度上说，一棵树上的任何节点（不包括根节点）都称为根的子孙节点。

祖先节点：若B是A的子孙节点，那么A即为B的祖先节点。

节点的层数：从根节点开始计算，根层数为 0，其余节点的层数为其双亲的层数加 1。例如图 7-2 中，节点H的层数为 3。

树的高度或深度：一棵树中所有节点层数的最大值。特别地，空树的高度为–1。

在画一棵树时，每个元素都被一个节点所代替。根在上，子树在下。根与子树之间有一种直接的联系，这种联系在 C++中通常使用指针实现。同样地，每一棵子树也是根在上，其子树在下。在一棵树中，边连接一个元素及其子节点。这就是前面曾经使用过的树的逻辑表示。除此之外表示树的方法还有很多，例如如图 7-3 所示是采用文氏图的方法来表示图 7-2 中的树形结构。

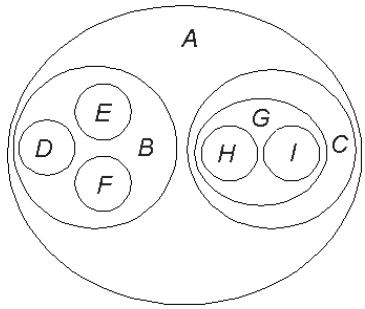


图 7-3　树的文氏图表示

7.1.3　树的抽象

因为树是节点的集合，因此组成树的“基本单元”是节点，在给出树类（Tree）的定义之前这里先给出了节点类（TreeNode）的定义。节点类主要是由 3 个成员变量组成的，它们是 m_value、leftChild 和 rightSibling，这些变量在节点之间起到了存储数据以及建立联系的作用。理解这种定义方式的关键在于理解树的递归本质。变量 m_value 代表着该节点中存储的数值；leftChild 表示该节点的第一个子节点，即最左子节点，它用于建立节点间的层次关系；rightSibling 表示与该节点最邻近的右兄弟节点。变量 leftChild 和 rightSibling 可以构成一种递归关系，并通过这种关系来构建树结构。

下面给出树的抽象数据类型定义。

```
template<class T> class Tree;
template<class T>
class TreeNode
{

friend class Tree<T>;

private:
    T m_value;
    TreeNode<T> * leftChild;       //第一个子节点，即最左子节点
    TreeNode<T> * rightSibling;    //该节点的右兄弟节点

public:

    TreeNode(const T&);            //构造函数
    ~TreeNode();                   //析构函数

    bool isLeaf();                 //是否是叶子节点

    T getValue();
    TreeNode<T>* getLeftChild();
    TreeNode<T>* getRightSibling();

    void setValue(T & value); //设置该节点值

    //设置该节点的最左子节点
    void setLeftChild(TreeNode<T>* pLeftChild);
    //设置该节点的右兄弟节点
```

```
    void setRightSibling(TreeNode<T>* pRightSibling);

};


template<class T>class Tree
{
    TreeNode<T>* root, * parent;

    int getParent(TreeNode<T>* root,TreeNode<T>* current);

public:

    Tree();
    Tree(TreeNode<T>* rt);     //构造函数，生成树的结构
    ~Tree();                   //析构函数

    TreeNode<T> * getRoot();   //返回树的根节点，若为空树，则返回 Null

    //返回节点 current 的父节点，如果节点 n 是根节点，则返回 Null
    TreeNode<T> * Parent(TreeNode<T> * current);

    //返回节点 t 的第一个子节点，若无则返回 Null
    TreeNode<T> * getFirstChild(TreeNode<T> * t);

    //返回节点 t 的下一个兄弟节点，若无则返回 Null
    TreeNode<T> * getNextSibling(TreeNode<T> * t);

    //在节点 t 下插入一个子节点 n
    void insertChild(TreeNode<T> * t, TreeNode<T> * n);

    void deleteSubTree(TreeNode<T> * t);    //删除以 t 为根节点的子树
    bool isEmpty();                         //判断是否为空树
};
```

7.2 花开二枝分外香——二叉树及相关算法

树形结构的具体形式有很多种，其中最常用的是二叉树。本节就来研究关于二叉树的一些内容。

7.2.1 二叉树的定义

小提示

二叉树是众多搜索操作实现的基础，后面我们在介绍平衡二叉树时，将进行更加详细的讨论。

二叉树（Binary Tree）作为一种具有特殊结构的树，本质上也是一个有限的节点的集合T，前面介绍的术语对于它同样适用。

组成二叉树的有限节点集合T或者为一空集，或者是由一个根节点和两个分别被称为左子树和右子树的互不相交的有限集合T_{left}和T_{right}共同组成。其中，T_{left}和T_{right}分别又是两棵独立的二叉树。从此定义中可以看出二叉树中蕴含着递归结构。因为T_{left}和T_{right}分别是两棵独立的二叉树，因此T_{left}和T_{right}也可以为空，所以二叉树中的每个节点最多有两个子女，即每个节点的度不超过 2，且兄弟节点之间有左右子树之分，次序不能颠倒。

如图 7-4 所示为一棵二叉树的例子。其中A是根节点，节点B、D、F组成了该二叉树的左子树，在图中用同样的颜色标出。节点C、E、G、H组成了该二叉树的右子树，在图中用同样的颜色标出。此外，B、D、F节点组成的树也是一个二叉树，其中B是根节点，它的左子树为D、F节点组成的二叉树，右子树为空。

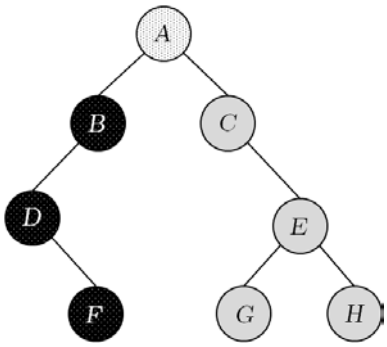


图 7-4　二叉树

在介绍二叉树性质之前先介绍两种特殊的二叉树。

（1）满二叉树（Full BinaryTree）

一棵深度为k且有$2^{k+1}-1$个节点的二叉树称为满二叉树。满二叉树具有如下特点。

- 每一层上的节点数都达到最大值。换言之，对于给定的高度，它是具有最多节点数的二叉树。
- 满二叉树中不存在度数为 1 的节点，即除了底层的节点度数为 0 外，其他节点的度数都为 2。满二叉树的每个分支节点均有两棵高度相同的子树，且树叶都在最下面一层上。

（2）完全二叉树（Complete BinaryTree）

若一棵二叉树至多只有最下面的两层上节点的度数可以小于 2，其余各层节点的度数都等于 2，并且最下面一层上的节点都集中在该层最左边的若干位置上，则此二叉树称为完全二叉树。如图 7-5 所示即为一棵完全二叉树。

完全二叉树具有如下特点。

- 叶子节点仅在层次最大的两层出现。
- 对于任意节点，如果其右子树的高度为 h，那么它的左子树的高度只能是h或者$h+1$。

由定义可知，满二叉树是完全二叉树，完全二叉树不一定是满二叉树。在满二叉树的最下面一层上，从最右边开始连续删去若干节点后得到的二叉树仍然是一棵完全二叉树。在完全二叉树中，若某个节点没有左孩子，则它一定没有右孩子，即该节点必是叶子节点。

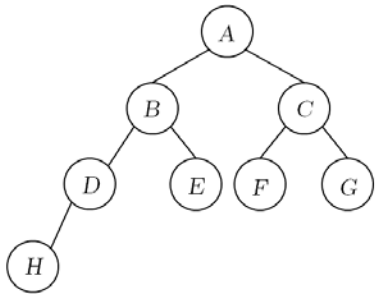


图 7-5　完全二叉树

7.2.2　二叉树的性质

二叉树结构严整，排列规则，因此也具有一些特有的良好性质，其中主要包括如下一些重要性质。

性质 1：对于非空二叉树，其第i层上的最大节点数目为2^i，其中$i \geqslant 0$。

证明：这里使用数学归纳法对此性质进行证明。

归纳基础：当$i = 0$时，有$2^0 = 1$个节点。因为第0层上只有一个根节点，所以命题成立。

归纳假设：假设对所有的j命题成立，即第j层上至多有2^j个节点，其中$0 \leqslant j \leqslant i$。

归纳步骤：根据归纳假设，第j层上至多有2^j个节点。由于二叉树的每个节点至多有两个孩子，故第$j+1$层上的节点数至多是第j层上的最大节点数的2倍。即得到$2 \times 2^j = 2^{j+1}$个节点，故命题成立。

性质 2：深度为k的二叉树至多有$2^{k+1} - 1$个节点，其中$k \geqslant -1$。

证明：易知当$k = -1$时，$2^{-1+1} - 1 = 0$，即此时该树中没有节点，表示该二叉树是一个空树，所以结论成立。当$k \geqslant 0$时，表示该二叉树是一个非空二叉树。又知在具有相同深度的二叉树中，仅当每一层都含有最大节点数时，其树中节点数最多。利用性质1可得，深度为k的二叉树的节点数至多为：

$$2^0 + 2^1 + \cdots + 2^k = \sum_{i=0}^{k} 2^i = 2^{k+1} - 1$$

故命题正确。

性质 3：在任意一棵非空二叉树中，若终端节点的个数为n_0，度为2的节点数为n_2，则$n_0 = n_2 + 1$。

证明：因二叉树中所有节点的度数均不大于2，所以节点总数（记为n）应等于0度节点数（记为n_0）、1度节点数（记为n_1）和2度节点数（记为n_2）之和，即$n = n_0 + n_1 + n_2$。另外，1度节点有一个孩子，2度节点有两个孩子，故二叉树中孩子节点总数是$n_1 + 2n_2$。树中只有根节点不是任何节点的孩子，故二叉树中的节点总数又可表示为$n = n_1 +$

$2n_2+1$。综上所述易得 $n_0=n_2+1$，故命题成立。

性质 4：具有n个节点的完全二叉树的深度为$k=\lceil\log_2(n+1)\rceil-1$。

证明：设所求完全二叉树的深度为k。由完全二叉树定义可知深度为k的完全二叉树的前$k-1$层是深度为$k-1$的满二叉树，一共有$2k-1$个节点。由于完全二叉树深度为k，故第k层上还有若干个节点，因此该完全二叉树的节点个数$n>2k-1$。另一方面，由性质2可得$n\leqslant 2k+1-1$，即$2k-1<n\leqslant 2k+1-1$。由此可推出$2k<n+1\leqslant 2k+1$，取对数后有$k<\log_2(n+1)\leqslant k+1$，易得$k+1=\lceil\log_2(n+1)\rceil$，故结论得证。

最后两个性质，我们仅给出结论，不做具体证明。

性质 5：如将一棵有n个节点的完全二叉树自上（根节点）向下（叶节点），同一层自左向右连续赋予节点编号$0,1,2,\cdots,n-1$，则对于任意下标为i的节点有：

（1）若$i=0$，则i是根节点，无父节点；若$i>0$，则i的父节点下标为$\lfloor(i-1)/2\rfloor$。

（2）若$2\times i+1<n$，则i的左子女为$2\times i+1$。

（3）若$2\times i+2<n$，则i的右子女为$2\times i+2$。

（4）若i为偶数，且$i\neq 0$，则其左兄弟为$i-1$。

（5）若i为奇数，且$i\neq n-1$，则其右兄弟为$i+1$。

（6）节点i所在的层次为$\lfloor\log_2(i+1)\rfloor$ 。

性质 6：在满二叉树中，叶子节点的个数比分支节点个数多1。

7.2.3　二叉树的实现

二叉树有两种常用的实现方式，一种是顺序方式，另一种是链式方式。顺序方式的实现是基于数组完成的，而链式方式的实现是基于链表完成的。

通常如果树形结构的规模和形态在使用过程中是固定的，那么可以考虑采用顺序方式实现。在用数组存储树形结构时，要注意必须按一定次序存储树中的节点数据，这样才能有效地维护元素间的关系，体现出树形结构的特点。

首先来讨论比较简单的情况——完全二叉树。假设有一棵完全二叉树如图 7-6 所示。如果希望使用数组对它进行存储。那么所遵循的规则是先对它的所有节点按照层次次序自顶向下、自左向右顺序编号为$0,1,2,\cdots,n-1$。以这些编号作为数组元素的下标将树中的节点线性地存放到一个数组中。

图 7-6 中的完全二叉树的数组表示结果如图 7-7 所示。当通过数组下标对树中元素进行访问的时候，可以通过 7.2.2 节中所给出的性质来推算出任意节点的父节点、子节点以及兄弟节点等的编号，进而找到这些节点。使用数组来存储完全二叉树简单高效，当树的规模和结构不会发生剧烈变化的情况下应当优先考虑这种方式。

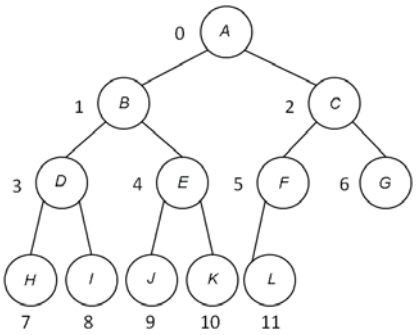


图 7-6　完全二叉树

0	1	2	3	4	5	6	7	8	9	10	11
A	B	C	D	E	F	G	H	I	J	K	L

图 7-7　完全二叉树的数组表示

下面将二叉树的数组表示推广到一般情况。同完全二叉树类似，先将二叉树自顶向下以层次遍历法对节点进行编码。但编码时需要注意如果遇到空子树，需要在假设该子树存在的情况下进行编码。就好像把一棵普通的二叉树看作是一个完全二叉树的子集，普通二叉树中节点的编号与其相应的完全二叉树中对应节点的编号是相同的。向数组中存放节点时，对于那些空缺的节点，在数组中标记为空，数组中对应位置是不存储任何元素的。这就是二叉树的数组表示。假设有如图 7-8 所示的一棵二叉树。

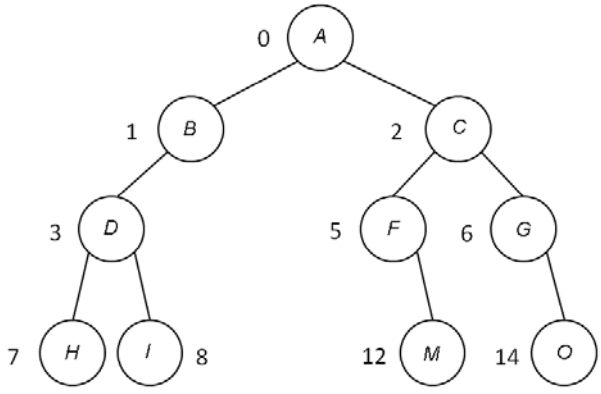


图 7-8　二叉树

可见图中编号为 1 的节点没有右子树，编号为 5 和 6 的节点没有左子树。对于这些缺失的节点，对应的数组中将会用一些“空”元素来表示。最终结果如图 7-9 所示。

0	1	2	3	4	5	6	7	8	9	10	1	12	13	14
A	B	C	D		F	G	H	I				M		O

图 7-9　二叉树的数组表示

可以发现图 7-9 中的数组出现了若干空缺的位置，这无疑造成了存储空间的浪费。但

这还不是最糟糕的情况。如果使用数组来存储一棵形如图 7-10 的“ 单支树 ”，读者可以自己尝试写一个数组来将其存放进去。不难发现数组中绝大部分的元素都是空缺的，这种资源的浪费将随着“ 单支树 ”深度的增长而越发惊人。

求职面经点拨

请试回答附录题库中的第 26 题。

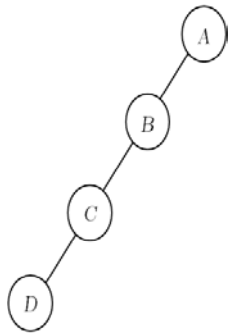


图 7-10　单支树

使用数组存储树形结构的优点在于实现简单，效率高，但是不足之处在于可能造成空间资源浪费。除此之外，基于数组的表达方式还有一个不方便之处。因为数组作为程序的一部分，它的存储空间是静态分配的，它的大小需要被提前设定。然而一开始如果分配过少的空间给数组，这有可能会造成溢出，如果一开始分配过多的空间给数组，这又造成了浪费。这对程序的灵活性也造成了一定的限制。为了摆脱这种局限性，可以考虑采用树形结构的链式存储方式。基于链表的树形结构存储方式将树中的每个节点看作是线性链表中的一个节点。每个节点都包含 3 个域，左孩子区域、右孩子区域及数据区域。节点之间通过左右孩子建立起类似索引的关系，从而清晰地体现出数据的层次性。图 7-11 给出了一个树形结构的链式节点的例子。其中，左孩子区域存储的是该节点的左子节点的指针，右孩子区域存储的是该节点的右子节点的指针。通过指针可以十分方便地构建起节点之间的关联关系。

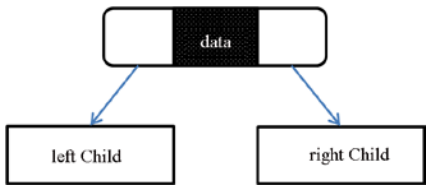


图 7-11　链式节点

链式节点是基于链表表示的树结构的基本单元。在了解了它的基本结构之后，现在来编程实现一个节点类（BinaryTreeNode）。它将用于辅助后面的二叉树类实现。下面给出 BinaryTreeNode.h 文件，其中包含了节点类的实现代码。

```
#ifndef BINARYTREENODE_H
#define BINARYTREENODE_H

#include "iostream"

using namespace std;
```

```
template <class Type>
class BinaryTreeNode
{
    Type m_data;                          // 存储该节点的数据
    BinaryTreeNode<Type> *m_leftChild; // 存储该节点的左孩子指针
    BinaryTreeNode<Type> *m_rightChild;// 存储该节点的右孩子指针

public:
    BinaryTreeNode(){m_leftChild = m_rightChild = NULL;};
    BinaryTreeNode(const Type &data,
                        BinaryTreeNode *leftChild = NULL,
                        BinaryTreeNode *rightChild = NULL)
    {
        m_data = data;
        m_leftChild = leftChild;
        m_rightChild = rightChild;
    };

    Type & GetData(){return m_data;};       //返回该节点的数据

    // 返回该节点的左孩子指针
    BinaryTreeNode<Type>* GetLeftChild(){return m_leftChild;};

    // 返回该节点的右孩子指针
    BinaryTreeNode<Type>* GetRightChild(){return m_rightChild;};

    // 设置该节点的数据
    void SetData(const Type & data){m_data = data;};

    // 设置该节点的左孩子指针域
    void SetLeftChild(BinaryTreeNode<Type> *leftChild)
        {m_leftChild = leftChild;};

    // 设置该节点的右孩子指针域
    void SetRightChild(BinaryTreeNode<Type> *rightChild)
        {m_rightChild = rightChild;};
};

#endif
```

对于这个类这里不做过多的说明，请读者留意其中注释说明的地方，它们将帮助读者理解这个类的构造和实现过程。

完成了节点类的设计，那么使用链表来组织树结构就不是一件困难的事了。整个二叉树被保存在一个线性的链表中，并通过一个指向根节点的表头指针来获得对于整个类的访问入口。链表中的每个节点都由 3 个区域组成，这点前面已经见过。如果一棵树的某个节点仅有左子节点，而无右子节点，那么它的左孩子区域存储的是该节点的左子节点的指针，右孩子区域存储的则是一个空指针，它不指向任何元素，反之亦然。如果一个节点是叶子节点，那么它的左孩子区域和右孩子区域就都为空。如图 7-12 所示为一棵二叉树的链表表示。不难发现，使用链表来表示二叉树可以很清晰地将树形结构的层次关系和其中节点的关联关系表现出来。

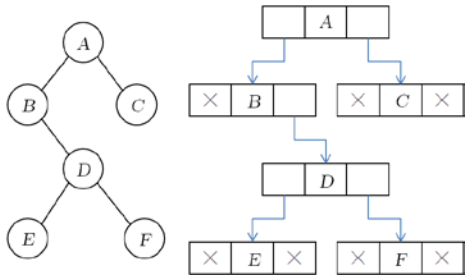


图 7-12　二叉树的链表表示

特别地，相对于用数组表示的二叉树，链表表示的二叉树具有很多优点。首先如果使用数组来表示二叉树，当二叉树为非完全二叉树时，数组中就会出现很多空白区域，这样会浪费很多空间。此外，一旦要进行插入和删除操作，很多情况下，数组的改动范围可能是很大的，这也给二叉树的操作实现带来了极大的不便。若运用链式存储，要进行节点的插入、删除、替换，所有的操作会被限定在模块化的范围中，从这个角度来看，链式存储是一种很好的选择。

完整实现二叉树类的代码可以从本书的在线支持资源中下载得到，限于篇幅此处不再具体列出。下面给出一段测试代码，它使用了二叉树类中的一些成员函数，并输出了包括层次遍历在内的一系列测试结果。

```
#include"BinaryTreeNode.h"
#include"BinaryTree.h"
#include"iostream"

using namespace std;

int main()
{
    BinaryTree<char> myBinTree('a');

    myBinTree.InsertLeftChild(myBinTree.GetRoot(), 'D');
    myBinTree.InsertRightChild(myBinTree.GetRoot(), 'G');

    myBinTree.InsertLeftChild(myBinTree.GetRoot(), 'B');
    myBinTree.InsertRightChild(myBinTree.GetRoot()->GetLeftChild(), 'E');

    myBinTree.InsertRightChild(myBinTree.GetRoot(), 'C');
    myBinTree.InsertLeftChild(myBinTree.GetRoot()->GetRightChild(), 'F');


    cout << "该二叉树是否为空? : " << myBinTree.IsEmpty() << endl;
    cout << "该二叉树的根节点中储存的数据是: "
        << myBinTree.Retrieve(myBinTree.GetRoot());
    cout << endl << "将二叉树的根节点中的数据赋值为 A!";
    myBinTree.Assign(myBinTree.GetRoot(), 'A');
    cout << "当前二叉树的根节点中储存的数据是: "
        << myBinTree.Retrieve(myBinTree.GetRoot()) << endl;

    cout << "层次遍历结果如下: " << endl;
    myBinTree.LevelOrderTraverse(myBinTree.GetRoot());
```

```
    cout << endl;
    return 0;
}
```

完成编码后，编译并运行程序即可。

上述二叉树类的实现过程中实际存在一个问题。那就是已知某个节点，希望求它的父节点的过程似乎有些复杂。可以看到代码中采用了一种递归的处理方式，通过前面的学习读者应当知道递归的代价是比较高的。特别当要确定父节点的节点处于树中很深的位置时，这个递归实现方式显然不是一个理想的选择。但是如果链节点中仅仅保存有该节点的左右子节点指针，那么向上回溯其父节点是不可能的，唯有从树的根节点进入逐层搜索。但这样效率又很低，如何解决这个矛盾呢？读者应该注意到了，这个情况似乎前面曾经遇到过。在研究链表的遍历时，我们遇到了几乎同样的困难。当在一个单向链表中寻找某个节点的后继节点十分容易，但如果是寻找它的前驱节点就要复杂得多了。为了解决这个矛盾，我们设计了循环链表。读者或许已经从中得到了启示。鉴于这种思想，我们可以采用一种“有来有往”的方式来设计链节点的结构。即用于存储树结构中节点的链节点不仅储存有该节点左右子节点的指针，同时存储着指向该节点父节点的指针。如此，获得一个节点的父节点的复杂度就变成了$O(1)$。

如图 7-13 所示，这里采用一种被称为“三叉链表”的结构来改造前面图 7-14 所存储的树形结构。这里链表中的每个链接点都包含 4 个区域，其中 1 个是数据区，另外 3 个都是指针区，存储的指针分别指向该节点的左子节点、右子节点和父节点，因此被称为“三叉”链表。对于根节点，它的父节点区中存储的是一个不指向任何地址的空指针。除了根节点以外，其他所有节点的父节点区都不能为空。左子节点区和右子节点区的情况与前面所述的情况无异，这里不再赘述。

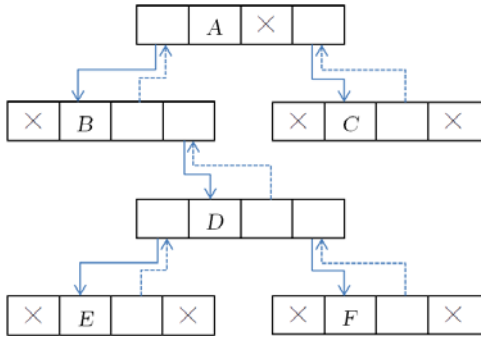


图 7-13　二叉树的三叉链表表示

读者不妨尝试改造一下前面的类，即采用三叉链表形式来表示树结构。限于篇幅，这里不再给出其实现代码。

算法

二叉树有 3 种遍历算法，即前序遍历、中序遍历和后序遍历。

7.2.4　二叉树的遍历算法

对于任何一种数据结构，都会规定一种遍历方法，在线性数据结构中，一般会根据它们的“天然”顺序结构进行遍历，而对于二叉树，则需要按照某种搜索路径访问二叉

树中的所有节点，使得每个节点被访问一次且仅被访问一次，这就是所谓的二叉树遍历。这里访问的含义就是指在一定权限下对节点进行的操作，比如说读取数据、修改数据等。

对二叉树进行遍历的 3 种最常用方案是前序、中序和后序遍历。如果使用*L*、*R*和*V*来分别代表遍历左子树、遍历右子树和访问该节点这 3 种操作，那么*VLR*代表前序遍历，*LVR*代表中序遍历，*LRV*则代表后续遍历。

在这 3 种方法中，相同的一点是每个节点的左子树在其右子树之前遍历。这 3 种遍历的区别在于对同一个节点在不同时刻进行访问。前序遍历时，中间节点是在其左右子树被访问之前进行访问的；中序遍历时，首先递归访问左子树，然后访问子树的根节点，最后访问右子树。在后序遍历时，当左右子树均访问完之后才访问子树的根节点。

下面对这 3 种遍历方法的原理及实现分别进行介绍。

1．前序遍历

前序遍历又称为先根遍历。其访问规则是如果二叉树为空则进行空操作，否则首先访问根节点，然后前序遍历左子树，最后前序遍历右子树。

如图 7-14 所示，若我们对图中的树进行前序遍历，则前序遍历从根节点 *A* 开始，遇到 *A* 后立即进行访问。然后查找 *A* 的左子树即以 *B* 为根的子树，随后立即访问节点 *B*。然后再查找 *B* 的左子树即以 *D* 为根的子树，随即访问节点 *D*。由于 *D* 没有子树，所以进行空操作，并回退到以 *B* 为根的子树，查找 *B* 的右子树，于是访问节点 E。再次回退到 *A*，并访问 *A* 的右子树，即以 *C* 为根的子树。这时访问节点 *C*。查找节点 *C* 的左子树即以 *F* 为根的子树，随即访问节点 *F*。由于 F 没有子树，所以进行空操作，并回退到以 *C* 为根的子树，查找 *C* 的右子树，于是访问节点 *G*。至此，二叉树的前序遍历完成，最终得到的遍历结果是 *ABDECFG*。

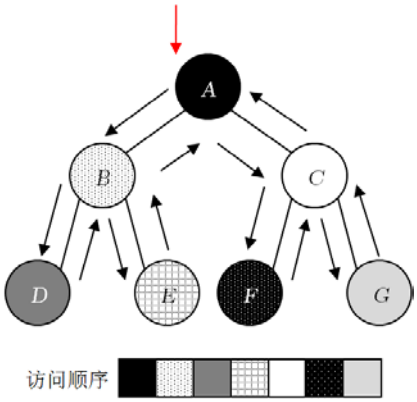


图 7-14　二叉树的前序遍历

下面来完成 7.2.3 节中二叉树类的 PreOrderTraverse 方法，代码清单如下。

```
template<class T>
void BinaryTree<T>::PreOrderTraverse(BinaryTreeNode<T> *root) const
{
   if (NULL != root)
   {
      cout << root->GetData();
      PreOrderTraverse(root->GetLeftChild());
      PreOrderTraverse(root->GetRightChild());
    }
}
```

除了这种递归的实现方法，还可以考虑使用栈结构来完成二叉树的前序遍历。基本原理是在压入一个元素后先进行访问，随即将其左子树压栈，当访问的节点不存在左子树时，退栈对右子树进行访问。直到栈为空，也就是所有节点访问完成。下面给出该方法实现的前序遍历源代码。

```
template<class T>
void BinaryTree<T>::PreOrderTraverse(BinaryTreeNode<T> *root)   const
{
    stack<BinaryTreeNode<T>*> s;
    BinaryTreeNode<T> *p=root;
    while(!s.empty()||p!=NULL)
    {
        while(p)
        {
            s.push(p);
            cout<<p->GetData();
            p=p->GetLeftChild();
        }
            p=s.top();
            s.pop();
            p=p->GetRightChild();
    }
}
```

2．中序遍历

中序遍历又称为中根遍历。其访问规则是如果二叉树为空则进行空操作，否则首先中序遍历左子树，然后访问根节点，最后中序遍历右子树。

如图 7-15 所示，若对图中的树进行中序遍历，则中序遍历从根节点 *A* 开始，但是并没有直接访问，而是首先访问 *A* 的左子树 *B*，遇到以 *B* 为根的子树后也没有直接访问 *B*，而是再次查找 *B* 的左子树，即以 *D* 为根的子树。因为 *D* 没有左右子树，所以访问 *D*。然后回退到以 *B* 为根的子树，访问 *B*，并查找 *B* 的右子树，即以 *E* 为根的子树。因为 *E* 没有左右子树，所以访问 *E*。访问完成后，回退到 *A*，因为 *A* 的左子树已经访问完毕，所以这时访问节点 *A*。然后按中序遍历的规则访问 *A* 的右子树。最终得到的遍历结果是 *DBEAFCG*。

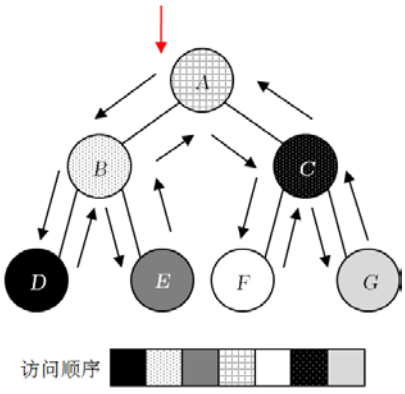


图 7-15　二叉树的中序遍历

下面使用递归的方法完成 7.2.3 节中二叉树类的 InOrderTraverse 方法, 代码清单如下。

```
template<class T>
void BinaryTree<T>::InOrderTraverse(BinaryTreeNode<T> *root) const
{
   if (NULL != root)
   {
      InOrderTraverse(root->GetLeftChild());
      cout << root->GetData();
      InOrderTraverse(root->GetRightChild());
   }
}
```

同样，除了这种递归的实现方法，还可以考虑使用栈结构来完成二叉树的中序遍历。下面给出第 2 种实现方法的源代码，请读者注意它与前序遍历在实现上的差异。

```
template<class T>
void BinaryTree<T>::InOrderTraverse(BinaryTreeNode<T> *root) const
{
    stack<BinaryTreeNode<T>*> s;
    BinaryTreeNode<T> *p=root;
    while(!s.empty()||p!=NULL)
    {
        while(p)
        {
            s.push(p);
            p=p->GetLeftChild();
        }

        p=s.top();
        s.pop();
        cout<<p->GetData();
        p=p->GetRightChild();
    }
}
```

3．后序遍历

后序遍历又称为后根遍历。其访问规则是如果二叉树为空则进行空操作，否则首先后序遍历左子树，然后后序遍历右子树，最后访问根节点。

如图 7-16 所示，对图中的树进行后序遍历，节点的访问过程已用颜色序列标出，这里

不再赘言。最终得到的遍历结果是 *DEBFGCA*。

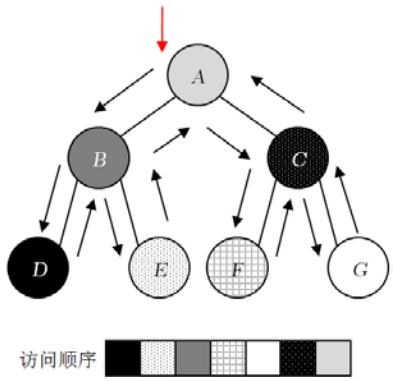


图 7-16　二叉树的后序遍历

使用递归方法完成 7.2.3 节中二叉树类的 PostOrderTraverse 方法，其代码清单如下。

```
template<class T>
void BinaryTree<T>::PostOrderTraverse(BinaryTreeNode<T> *root) const
{
   if (NULL != root)
   {
      PostOrderTraverse(root->GetLeftChild());
      PostOrderTraverse(root->GetRightChild());

      cout << root->GetData();
   }
}
```

同样，除了这种递归的实现方法，还可以考虑使用栈结构来完成二叉树的后序遍历。限于篇幅这里不再给出实现代码。有兴趣的读者可以参照前面给出的 PreOrderTraverse 和 InOrderTraverse 函数的非递归实现方法来自己实现二叉树后序遍历的非递归方法。

在本书提供的在线资源中，还包含有二叉树层次遍历的实现代码。该方法正像它的名字所表述的那样，对二叉树的每一层进行从左向右的顺序遍历。读者应当注意到层次遍历的实现所利用的辅助数据结构是队列。这也是该遍历方式与本节说述的 3 种方法在实现上的明显差异。在逐层遍历过程中，按从顶层到底层的次序访问树中元素，而在同一层中，从左到右进行访问。前中后这 3 种遍历方案的路线完全一样，只是访问时间不同。由于两类访问策略存在本质上的区别，因此前中后遍历采用栈作为辅助数据结构，而层次遍历采用了队列。这是值得读者注意的地方。

7.2.5　二叉树线索化算法

前面几节对二叉树的存储和遍历做了介绍。如果利用曾经讲过的几种遍历方法对二叉树进行遍历，那么树中的所有节点就可以按照一定的次序被放进一个有序的线性表中。根据本书前几章中所论述的内容，读者应当知道，如果希望得到某个线性表中任意元素的前驱或者后继通常采取的办法就是从头开始遍历该表。但是如果希望更快地得到表中任意元素的前驱或者后继呢？

在计算机问题解决的方法设计上，研究者经常需要在两个方面进行平衡和折中，即时间代价和空间代价。通常以牺牲时间为代价可以换得空间上的优化，反之亦然。举一个简单的例子。比如要计算两个大整数的乘法，若是因为空间资源相对宝贵，因此希望能够尽量节省空间消耗，方法通常就是按照乘法规则对两个整数进行运算。但是如果空间资源相对没有时间那么宝贵，不妨采用的一个方法就是预先把答案保存在存储器上，便可免去计算的时间，直接输出答案。

基于这种思想，为了能够更快地得到表中任意元素的前驱或者后继，可通过将每个元素的前驱和后继都存储起来的方式来避免遍历，这样就实现了时间上的高效。为此需对图 7-11 中的二叉链式节点进行改造，增加一个前驱指针域（Pred）和一个后继指针域（Succ）。前驱指针域指向在某种顺序下的前驱节点，而后继指针域指向在某种顺序下的后继节点。改造后的链节点如图 7-17 所示。

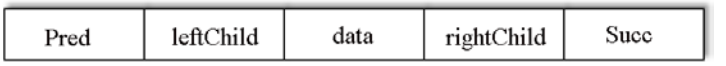


图 7-17　增加了 Pred 指针和 Succ 指针的链节点

这样做已经达到降低时间开销的目的了。但是显然也浪费了许多宝贵的空间，是否可以做一些优化呢？当以二叉链表作为二叉树的存储结构时，因为只能找到节点的左右孩子的信息，而不能获得节点在任一序列下的前驱与后继信息，这种信息只有在遍历的动态过程中才能得到，为了能保存所需的信息，考虑增加前驱指针域和后继指针域。但又希望能够尽量减少空间开销，于是人们设计使用前驱（后继）标志域来替代原来的前驱（后继）指针域，而使前驱（后继）指针域和左（右）孩子域共享一块区域。以这种结构构成的二叉链表作为二叉树的存储结构，叫作线索链表，其中指向节点前驱与后继的指针叫作“线索”，即 Pred 指针和 Succ 指针。加上线索的二叉树称为线索二叉树，给二叉树加线索的过程称为线索化（穿线）。根据线索性质的不同，线索二叉树可分为前序线索二叉树、中序线索二叉树和后序线索二叉树 3 种。

前面的描述可能有些模糊，读者容易搞不清楚如何使前驱（后继）指针域和左（右）孩子域共享一块区域，前驱（后继）标志域又如何起作用。先来回答第一个问题。如图 7-11 所示，每个二叉链式节点都有两个指针域，即左孩子域和右孩子域，所以一个有n个节点的二叉树，应当有$2n$个指针。这些指针是否全部被使用了呢？当然没有。存储树形结构的二叉链表中存在很多空指针域。到底空了多少呢？对于不同的二叉树情况是不一样的。但估算出存在空指针域最少的情况还是很可能的。存在空指针域最少的情况就是当二叉树是一棵满二叉树的时候。下面来考虑一棵有n个节点的满二叉树中存在多少个空指针域。

一棵有n个节点的满二叉树的二叉链表中应当有$2n$个指针。设满二叉树的深度为k，其中$k \geqslant -1$。那么根据等比数列求和公式可以推导出n与k的关系如下。

$$n = 2^0 + 2^1 + \cdots + 2^k = 2^{k+1} - 1$$

将上述等式左右两边同时乘以2得到$2n = 2 \times (2^{k+1} - 1)$。这个等式表示具有$n$个节点

的满二叉树中存在$2 \times (2^{k+1} - 1)$个指针域。对于满二叉树来说，空指针域仅存在于它的叶子节点中。一个深度为k的满二叉树应当具有2^k个叶子节点，因此也就具有2^{k+1}个空指针域。那么非空指针域的数目即为$2 \times (2^{k+1} - 1) - 2^{k+1} = 2^{k+1} - 2$。由此易得用来存储满二叉树的具有$2n$个指针域的二叉链表中只用了$n - 1$个指针域，有$n + 1$个指针是空的。二叉树线索化就是要充分利用这些空指针域。

一般约定，对于空的左孩子指针域，用来存放指向遍历序列的前驱节点的指针；对于空的右孩子指针域，就用来存放指向遍历序列的后继节点的指针。为了区分指针域中存放的到底是指向树形结构中子节点的指针还是指向有序序列中前驱（或后继）节点的指针，就必须用到标志域。在每个节点中都设置有两个标志域leftThread和rightThread。如果leftThread = 0，表示leftThread域指示的是节点的左孩子，如果leftThread = 1，则表示leftThread域指示的是节点的前驱。如果rightThread = 0，表示rightThread域指示的是节点的右孩子，如果rightThread = 1，那么表示rightThread域指示的是节点的后驱。由于每个标志域仅仅占用一个二进制位，所以每个节点所需要的存储空间就节省了很多。最终线索化二叉树的链节点如图 7-18 所示。

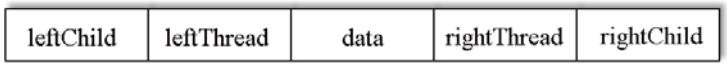


图 7-18　线索化二叉树的链节点

对二叉树进行线索化，实质上就是将每个节点中空的左右孩子指针域分别修改为指向其中序序列的前驱、后继节点。线索化算法可以表述为：首先因为每个节点均可能有空指针需要线索化，所以先对二叉树进行遍历。其次，对于每个节点进行线索化操作，即如果左子树为空，左标志置 1，左孩子指针改为指向其前驱。如果右子树为空，右标志置1，右孩子指针改为指向其后继。特别地，对每个节点进行线索化时，应知道每个节点的前驱和后继节点的地址。当线索化某一节点时，应当使用一个指针变量 pre 来记录当前节点的地址，当遍历到下一个节点时，如果下一节点需要记录前驱节点的指针值，那么就可以从指针变量 pre 中获得。而对于某一节点的后继节点指针值，在线索化该节点时是未知的，必须按照遍历次序向后寻找。也就是说，对p所指节点为当前节点的情况下，完成的操作应有两个，即对p所指节点的前驱节点（由 pre 指示）进行后继线索化，以及对p所指节点进行前驱线索化。

读者可能会想到线索化二叉树中不是所有节点都能够直接找到其后继（或前驱）的。如果在节点的rightChild中存放的是后继线索，那么就可以通过后继线索直接找到该节点的直接后继；否则，当rightChild中存放的是右子女指针时，该节点在某种顺序下的后继就是无法直接获得的，这时就需要通过一定的运算才能找到该节点的后继。表 7-1 给出了在中序线索二叉树中获得节点前驱和后继的规则。

表 7-1　在中序线索二叉树中获得节点前驱和后继

查找后继				查找前驱			
当前节点		rightThread		当前节点		leftThread	
		0	1			0	1
rightChild	根节点	/	无后继	leftChild	根节点	/	无前驱
	非根节点	后继为当前节点右子树的中序下第一个节点	后继为右子女		非根节点	前驱为当前节点左子树的中序下最后一个节点	前驱为左子女

相应地在前序线索二叉树和后序线索二叉树中也能构建出这种规则，这里不再给出，有兴趣的读者可以自己尝试着推导。

7.3　合抱之木，生于毫末——从树到森林

7.2 节介绍了二叉树，本节将对二叉树进一步泛化，介绍树这种数据结构。二叉树是树的一种特例，由树结构所表示的模型能够反映更加普遍的情况。此外，本节还将介绍有关森林的一些概念。

7.3.1　树的存储表示

树的存储表示方法有很多种，这里介绍其中比较常用的几种。

1．双亲节点数组表示法

由树的定义可以知道，树中的每个节点（除根节点以外）都有唯一的一个父节点。根据这一特性，可用一组连续的存储空间（一维数组）存储树中的各个节点，数组中的一个元素表示树中的一个节点，数组元素为结构体类型，其中包括节点本身的信息（data 域）以及节点的双亲节点在数组中的序号（parent 域），如果 parent 域的值为–1 则表示该节点无双亲节点，即该节点是一个根节点。树的这种存储方法称为双亲节点数组表示法。如图 7-19 所示的一棵树可以采用双亲节点数组表示法存储为如表 7-2 所示的情况。

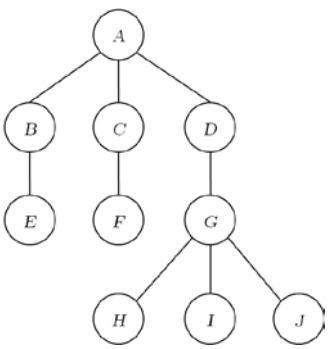


图 7-19　树

表 7-2　树的双亲节点数组表示法

index	data	parent
0	A	–1
1	B	0
2	C	0
3	D	0
4	E	1
5	F	2
6	G	3
7	H	6
8	I	6
9	J	6

2．孩子节点多重链表表示法

由于树中每个节点都有零个或多个孩子节点，因此，可以令每个节点包括一个节点信息域和多个指针域，每个指针域指向该节点的一个孩子节点，并通过各个指针域值反映出树中各节点之间的逻辑关系。如果，某些节点的孩子节点是叶子节点，那么无须通过指针指向这些节点，而可以直接将该节点的所有是孩子节点的子节点直接链入以该节点作为表头的链表中。在这种表示法中，树中每个节点有多个指针域，形成了多条链表，所以这种方法常被称为多重链表法。树的多重链式表示是一种非常有效存储树形结构的方法。这种表示方法的出发点是通过树形结构的层次特点，将节点串连在若干个链表内，然后再将这若干个链表进行组合。如图 7-19 所示的树可以采用多重链表表示法存储为如图 7-20 所示的情况。

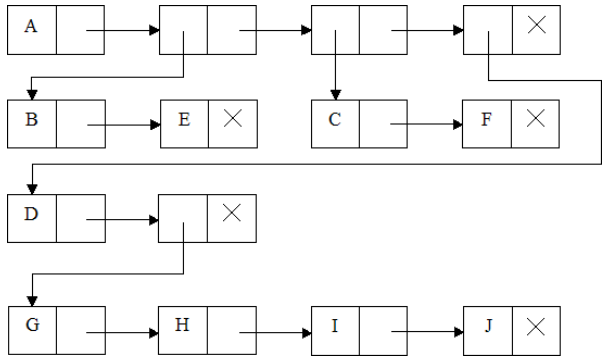


图 7-20　树的多重链式表示

3．孩子节点链表表示法

该表示方法为树的每个节点建立一个孩子链表。孩子链表表示法是将树按如图 7-21 所示的形式存储，以如图 7-19 所示的树为例。主体是一个与节点个数一样大小的一维数组，数组的每一个元素都由两个域组成，一个域用来存放节点信息，另一个用来存放指针，该指针指向由该节点孩子组成的单链表的首位置。单链表的结构也由两个域组

成，一个存放孩子节点在一维数组中的序号，另一个是指针域，指向下一个孩子。

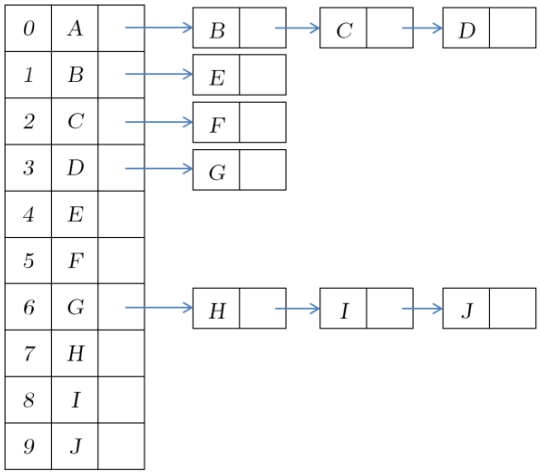


图 7-21　树的孩子节点链表表示法

4．左子女—右兄弟表示法

左子女—右兄弟表示法是目前已知的最节省存储空间的树的存储表示方法。它是一种基于递归的定义方式。本章前面所给的树的抽象数据类型就是采用此方法来定义树形结构的。该方法的基本原理是在每个节点中提供 3 个域，这 3 个域除了存储该节点自身信息的信息域外，还包括两个分别指向该节点的第一个孩子节点（即最左子节点）和该节点的最邻近兄弟节点（即右兄弟）的指针。图 7-22 给出了该方式所使用的节点结构样例。

图 7-22　左子女—右兄弟表示法的节点单元

图 7-19 中，树的根节点是A，它没有兄弟，所以它的rightSibling域为空。A有3个子节点B、C和D，因此它的leftChild域存储的是它的第一个子节点的地址，即节点B的地址。节点B有兄弟节点，也有子节点，它的rightSibling域存储的是其最邻近兄弟节点的地址，即C节点的地址，它的leftChild域存储的是节点E的地址。而节点E既没有兄弟节点，也没有子节点，所以它的rightSibling域和leftChild域都为空。依照此规则可以很方便地建立起树形结构的存储模式。

理解该表示方法的关键在于理解它的递归本质。为了帮助读者理解为什么仅通过每个节点的 3 个域就能实现整棵树的存储，这里来试着按这种结构从根节点进入遍历图 7-19 中的树。因根节点A没有右兄弟节点，仅有左子节点，故遍历过程访问根节点A的左子节点B。这时B既有兄弟节点，也有子节点，先来访问它的子节点E。因为E是叶子节点，既没有兄弟节点，也没有子节点，所以遍历过程回退到节点B，并访问它的右兄弟节点C。节点C既有兄弟节点，也有子节点，先来访问它的子节点F。因为F是

叶子节点，没有兄弟节点，也没有子节点，所以遍历过程回退到节点C，并访问它的右兄弟节点D。节点D仅有左子节点，而没有右兄弟节点，所以访问它的左子节点F。节点F同样仅有左子节点，没有右兄弟节点，同样访问它的左子节点H。节点H没有左子节点，而仅有右兄弟节点，所以访问它的右兄弟节点I。节点I没有左子节点，仅有右兄弟节点，所以访问它的右兄弟节点J。J既没有右兄弟节点，也没有左子节点。至此，遍历结束。可见尽管左子女—右兄弟表示法的每个节点单元仅有3个域，但它们对于构建树形结构的复杂模型已经足够了。

7.3.2 树的实现

本章 7.1.3 节给出了树的抽象数据类型，该抽象数据类型其实是由节点类和树类这两个类声明组成的。树类是基于 7.3.1 节中介绍的左子女—右兄弟表示法给出的。本节介绍节点类和树类这两个类中部分函数的实现方法。下面首先给出节点类中部分主要成员函数的实现代码。因为这些函数的实现都非常简单，这里就不再做过多的解释了。

```
template<class T>
TreeNode<T>::TreeNode(const T& value)
{
    m_value=value;
    leftChild=rightSibling=NULL;
}

template<class T>
bool TreeNode<T>::isLeaf()
{
    if(NULL==leftChild)
        return true;
    else return false;
}

template<class T>
T TreeNode<T>::getValue()
{
    return m_value;
}

template<class T>
void TreeNode<T>::setValue(T & value)
{
    m_value = value;
}

template<class T>
TreeNode<T>* TreeNode<T>::getLeftChild()
{
    return leftChild;
}

template<class T>
TreeNode<T>* TreeNode<T>::getRightSibling()
{
    return rightSibling;
}
```

```
template<class T>
void TreeNode<T>::setLeftChild(TreeNode<T>* pLeftChild)
{
    leftChild = pLeftChild;
}

template<class T>
void TreeNode<T>::setRightSibling(TreeNode<T>* pRightSibling)
{
    rightSibling = pRightSibling;
}
```

树类中部分函数的实现代码如下。

```
template<class T>
Tree<T>::Tree()
{
    root = NULL;
    parent = NULL;
}

template<class T>
Tree<T>::Tree(TreeNode<T>* rt)
{
    root = rt;
    parent = NULL;
}

template<class T>
TreeNode<T> * Tree<T>::getRoot()
{
    return root;
}

template<class T>
bool Tree<T>::isEmpty()
{
    if(root == NULL)
        return true;
    else
        return false;
}

template<class T>
int Tree<T>::getParent(TreeNode<T>* t,TreeNode<T>* p)
{
    TreeNode<T>* q = t->getLeftChild();

    while(q!=NULL && q!=p)
    {
        if( getParent(q, p) != 0 )
            return 2;

        q = q->getRightSibling();
    }

    if(q!=NULL && q==p)
    {
        parent = t;
        return 1;
    }
```

```
    else return 0;

}

template<class T>
TreeNode<T> * Tree<T>::Parent(TreeNode<T>* current)
{
    TreeNode<T>* pointer=current, *t;

    if(current==NULL||current==root)
    {
        current = NULL;
        return 0;
    }

    t = root;

    getParent(t, pointer);

    return parent;
}

template<class T>
TreeNode<T> * Tree<T>::getFirstChild(TreeNode<T> * t)
{
    return t->getLeftChild();
}

template<class T>
TreeNode<T> * Tree<T>::getNextSibling(TreeNode<T> * t)
{
    return t->getRightSibling();
}

template<class T>
void Tree<T>::insertChild(TreeNode<T> * t, TreeNode<T> * n)
{
    if(t->getLeftChild()==NULL)
        t->setLeftChild(n);
    else
    {
        TreeNode<T> * p=t->getLeftChild();
        while(p->getRightSibling()!=NULL)
            p=p->getRightSibling();
        p->setRightSibling(n);
    }
}
```

以上完成的成员函数中需要提醒读者注意的是以下几个。

- insertChild(TreeNode<T> * t, TreeNode<T> * n)

该函数在节点t下插入一个新的子节点n。如果节点t是叶子节点，那么就将节点n作为它的长子节点，即最左子节点。否则，如果节点t不是叶子节点，那么就将节点n作为它的最右子节点。

- Parent(TreeNode<T>* current)

成员变量 parent 中存储当前节点的父节点。公有函数 Parent 首先判断当前节点是否为

空或者是否为根节点，如果判断结果为真，那么表明当前节点没有父节点，所以返回空。否则调用私有函数 getParent。getParent 会更新 parent 的内容。最后函数 Parent 将更新后的 parent 返回。

- getParent(TreeNode<T>* t,TreeNode<T>* p)

私有成员函数 getParent 采用深度优先的策略对树进行遍历，一旦发现某节点是待寻找父节点的节点，那么就将其父节点赋给 parent，并通过这种方式更新 parent。

7.3.3　树与森林的遍历算法

算法

树的遍历算法主要有两种，即深度优先遍历和广度优先遍历。这与我们后面将要介绍的图的深度优先遍历与广度优先遍历是相通的。

遍历一棵树与遍历一棵二叉树意义相同，都是要以某种次序访问所有节点，二叉树因为有特定的结构，所以可以采用前中后 3 种遍历方法。二叉树的遍历方法与树的遍历方法是存在区别的，本节主要介绍树与森林的遍历问题，树的遍历是基于根的，而森林的遍历则是基于树的。

到目前为止森林还是一个陌生的概念。但就经验而言，对于大多数学习数据结构的人在理解了树的概念之后，很容易在头脑中形成森林的概念。数据结构中森林与树的关系与自然界中森林与树的关系是一致的，即森林是由树组成的集合。下面就给出森林的准确定义：森林（也称树林），是若干棵互不相交的树组成的集合。一棵树可以看成是一个特殊的森林。

下面先介绍树的遍历方法。总的来说，树的遍历方法有两种：深度优先遍历和广度优先遍历。关于深度优先遍历和广度优先遍历的概念在本书后面还会进行讲述，那时读者将会对它们有更深刻的理解，这里只针对树的遍历说明操作的具体方法和步骤。

1．深度优先遍历

深度优先遍历方式又可以分为两种具体的形式：先根遍历和后根遍历。通常树形结构中没有具体规定子树的先后次序，这里人为规定从左到右，子树的顺序依次为：第 1 棵子树、第 2 棵子树、第 3 棵子树……

先根遍历是一种将根的访问作为第一优先级的遍历方法。它的规则很简单，即如果树非空，则访问其根节点；然后从左到右先根遍历根的每棵子树。

如果采用先根遍历的方式遍历图 7-19 中的树，其遍历序列为 *ABECFDGHIJ*。遍历过程如图 7-23 所示。

后根遍历是一种将根的访问作为第二优先级的遍历方法。其规则是若树非空，则从左到右后根次序遍历根的每棵子树，然后访问根节点。

如果采用后根遍历的方式遍历图 7-19 中的树，其遍历序列为 *EBFCHIJGDA*。遍历过程如图 7-24 所示。

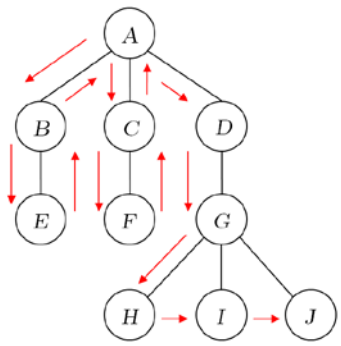


图 7-23　树的先根遍历

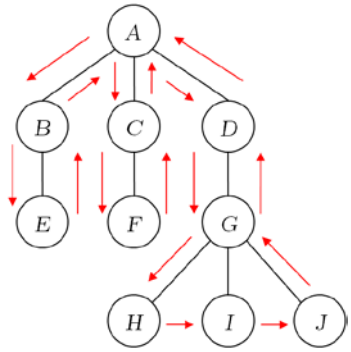


图 7-24 树的后根遍历

2．广度优先遍历

树的广度优先遍历与二叉树的分层遍历类似。可以认为二叉树的分层遍历是树的广度优先遍历在二叉树情况下的特例。广度优先遍历方式遵循的是分层访问原则。遍历首先访问树中深度为 0 的那一层中的节点，然后再访问深度为 1 的那一层中的节点……在访问每一层节点的时候，遵循的原则是从左向右。直到树中的所有节点都被访问过。如果采用广度优先遍历的方式遍历图 7-21 中的树，其遍历序列为 *ABCDEFGHIJ*。

下面先介绍森林的遍历方法。森林的遍历是基于树的遍历实现的，可选的遍历方式同样有深度优先和广度优先两种，但是由于后者很少被用到，因此这里仅介绍森林的深度优先遍历。森林的深度优先遍历可以采用先根遍历和后根遍历两种方式。

森林的先根遍历，又称为先序遍历，其规则如下：当森林非空时，访问森林的第 1 棵树的根节点，然后先根遍历第 1 棵树的子树森林（一棵树除去根节点剩余的部分就是一个子树森林），再以先根次序遍历其他树组成的森林。如果森林为空则执行空操作。

以如图 7-25 所示的森林为例，先访问以 *A* 为根的树，最后访问以 *F* 为根的树，先序遍历结果为 *ABCDEFGHIJ*。

森林的后根遍历规则为：当森林非空时，后根遍历森林中第 1 棵树除去根节点形成的子树森林，然后访问森林中第 1 棵树的根节点，再后根遍历其他树组成的森林。如果森林为空则执行空操作。以如图 7-25 所示的森林为例，其后根遍历结果为 *BCAEDHIJGF*。

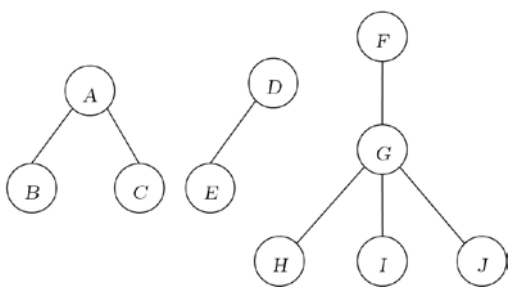


图 7-25　森林

7.3.4　森林与二叉树的转换

把简单的问题看得复杂，可以发现新的领域；把复杂的问题看得简单，可以发现新的规律。通常在较为特殊的情况下，事物的规律性更强，更容易被操作和掌握。因此，如果能够将一般的情况转化为较为特殊的情况，进而对较为特殊的情况进行研究，那么往往会使事物简单化。本节就研究森林、树与二叉树的转换问题。

1．树转换为二叉树

进行树与二叉树的转换的关键问题在于寻找树与二叉树之间的对应关系。由于树中每个节点可能有多个孩子，但二叉树中每个节点最多只能有两个孩子，所以二者之间的对应关系应当能够解决该矛盾。前面曾经介绍过树的左子女—右兄弟表示法，因此读者应当知道树中每个节点最多只有一个最左的孩子和一个右邻的兄弟，这就是二者转换操作实现的原理所在。通常树是无序的，但为避免发生混淆，这里约定树中每一个节点的孩子节点也按从左到右的次序。

将一棵树转换为二叉树的方法如下。

（1）树中所有相邻兄弟之间加一条连线。

（2）对树中的每个节点，只保留它与第 1 个孩子节点之间的连线，删去它与其他孩子节点之间的连线。

（3）以树的根节点为轴心，将整棵树顺时针转动一定的角度，使之结构层次分明。

下面来举例说明该转换方法的具体操作步骤。图 7-26 演示了将一棵树按上述方法转换为二叉树的过程。

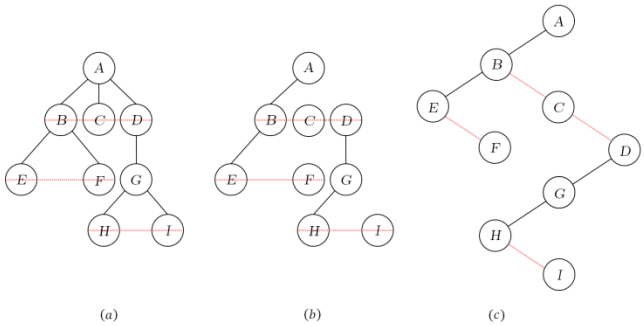


图 7-26　树与二叉树的转换

2．森林转化为二叉树

森林是若干棵树的集合，将森林中各棵树的根视为兄弟，每棵树都用二叉树表示，这样森林也同样可以用二叉树表示。森林到二叉树的转换方法如下。

（1）将森林中的每棵树转换成相应的二叉树。

（2）第 1 棵二叉树不动，从第 3 棵二叉树开始，依次把后一棵二叉树的根节点作为前

一棵二叉树根节点的右孩子，当所有二叉树连起来后，此时所得到的二叉树就是由森林转换得到的二叉树。

如图 7-27 所示为一个森林转换为一棵二叉树的具体过程。

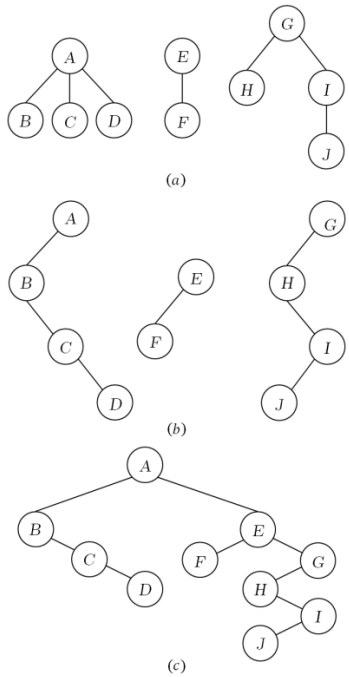


图 7-27　森林与二叉树的转换

3．二叉树转换为树和森林

可以依据二叉树的根节点有无右分支，将一棵二叉树还原为树或森林，具体方法如下。

（1）若某节点是其双亲的左孩子，则把该节点的右孩子，右孩子的右孩子，……，都与该节点的双亲节点用线连起来。

（2）删去原二叉树中所有的双亲节点与右孩子节点的连线（新添加的连线不删除）。

（3）整理由（1）、（2）两步所得到的树或森林，使之结构层次分明。

如图 7-28 所示为一棵二叉树转换为一个森林的具体过程。

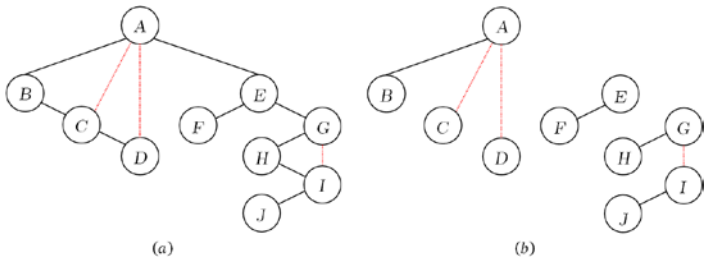


图 7-28　二叉树转换为森林

体现二叉树、树与森林转换意义的一个典型例子是它们的遍历顺序之间存在某种联系，具体可以总结为如下几点。

- 对树的二叉树表示进行先根次序遍历的结果与对应二叉树的前序遍历结果相同。
- 对树的二叉树表示进行后根次序遍历的结果与对应二叉树的中序遍历结果相同。
- 森林的先根次序遍历结果与其对应的二叉树的前序遍历所得到的结果相同。

限于篇幅这里不再给出上述命题的证明过程，但读者可以自己尝试验证一下它们的正确性。此外，请读者思考一个问题——森林的后根次序遍历结果与其对应的二叉树的后序遍历所得到的结果是否相同?

7.4　哈夫曼树——最优二叉树编码算法

在数据膨胀、信息爆炸的今天，数据压缩的意义不言而喻。要谈到数据压缩，就不能不提到哈夫曼编码。哈夫曼编码是一种常用的压缩编码方式。而哈夫曼树则是构造该编码时需要用到的一种树形数据结构。

7.4.1　哈夫曼编码

1948 年，美国数学家克劳德 · 艾尔伍德 · 香农（Claude Elwood Shannon）发表了著名的《通信的数学理论》一文，标志着信息论的诞生。3 年后，即 1951 年，同样是在香农曾经就读过的麻省理工学院，后来的美国计算机科学家大卫 · 阿伯特 · 哈夫曼彼时（David Albert Huffman）还在攻读博士学位。哈夫曼的导师罗伯特 · 马里奥 · 法诺（Robert Mario Fano）让学生们选择是完成学期报告还是参加期末考试。为了证明自己可以不用参加期末考试，哈夫曼最终选择完成学期报告，而这个看似不经意的举动最终造就了影响深远且意义非凡的哈夫曼编码。时至今日，哈夫曼编码仍然被广泛地应用于诸多领域。

法诺是著名的信息论专家，曾经和香农共同研究并提出香农-法诺编码。法诺给他的学生们布置的学期报告题目是，查找最有效的二进制编码。由于无法证明哪个已知编码是最有效的，哈夫曼放弃对已有编码的研究，转向新的探索，最终发现了基于有序频率二叉树编码的想法，并很快证明了这个方法是最有效的。哈夫曼使用自底向上的方法构建二叉树，避免了次优算法香农-法诺编码的最大弊端——自顶向下构建树。学生终于青出于蓝，超过了他的导师。最终在 1952 年，哈夫曼发表了阐释哈夫曼编码的论文，正式向世人介绍其发明的编码技术。

哈夫曼编码效率高，运算速度快，实现方式灵活，从 20 世纪 60 年代开始，哈夫曼编码在数据压缩领域得到了广泛的应用。而 20 世纪 80 年代初，哈夫曼编码又出现在 CP/M 和 DOS 系统中。即使在今天，许多知名的压缩工具和压缩算法里，依然可以见到哈夫曼编码的影子。例如，在 JPEG 图像压缩方式中，就用到了哈夫曼编码。

哈夫曼编码是一种不等长的编码，其基本原理是频繁使用的数据用较短的代码代替，而较少使用的数据则用较长的代码代替，这样就可以使报文中的编码数减少到最少，从而达到压缩的目的。在变长编码中，对出现概率大的信源符号赋予短码字，而对于出现概率小的信源符号赋予长码字。如果码字长度严格按照所对应符号出现概率大小的逆序排列，则编码结果平均码字长度一定小于任何其他排列方式。这就是哈夫曼提出的对统计独立信源能达到最小平均码长的编码方法，也就是最佳码。其最佳性可从理论上证明，这种编码方式具有即时性和唯一可译性。

在数据通信中，用二进制数给每个字符进行编码时不得不面对的一个问题是如何使电文总长最短且不产生二义性。哈夫曼树（Huffman Tree）可以用来解决这个问题。哈夫曼树是哈夫曼编码的基础，利用哈夫曼树可以构造哈夫曼编码。简而言之，哈夫曼树就是带权路径长度最小的二叉树，它通过将权值较大的外节点调整到离根节点较近的位置，从而得到最小路径长度。哈夫曼提出了一种构造哈夫曼树的算法，称为哈夫曼算法。根据字符出现频率，利用哈夫曼树可以构造一种不等长的二进制编码，并且构造所得的哈夫曼编码是一种最优前缀编码，它可以使编码后的报文长度最短，且保证任何一个字符的编码都不是同一字符集中另一个字符的编码的前缀。

哈夫曼树就是带权路径长度最小的二叉树，那么什么是带权路径长度呢？在具体介绍哈夫曼树的定义之前，先给出一些基本概念的定义。

- 节点的路径长度

从根节点到该节点的路径上的边数。

- 树的路径长度

树中每个叶子节点的路径长度之和。

- 节点的带权路径长度

定义为节点的路径长度与节点权值的乘积。

- 树的带权路径长度

树的带权路径长度（WPL，Weighted Path Length）是树中所有叶子节点的带权路径长度之和，即$\mathrm{WPL}(T)=\sum_{i=1}^{n}W_i l_i$，其中，$W_i$表示树中各叶子节点的权值，$l_i$表示叶子节点的路径长度。

下面举例说明上述各概念的意义。如图 7-29 所示为两棵二叉树，其中图(a)中二叉树节点6的路径长度为2，该树的路径长度为8；图(b)中二叉树节点6的路径长度为3，该节点的带权路径长度为18，该树的路径长度为9。这两棵二叉树的带权路径长度分别为：

$$\mathrm{WPL}_{(a)}=(1+2+4+6)\times 2=26$$

$$\mathrm{WPL}_{(b)}=1+2\times 2+(4+6)\times 3=35$$

设节点集合$K=\{k_1,k_2,\cdots,k_n\}$，并以$k_1,k_2,\cdots,k_n$为叶子节点构造二叉树，如果它们的权值分别为$w_1,w_2,\cdots,w_n$，那么称使得该二叉树的WPL值最小的二叉树为哈夫曼树或最优二叉树。

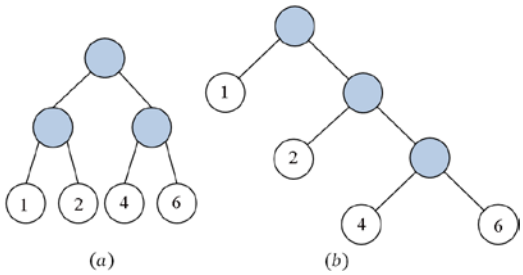


图 7-29　树的带权路径

哈夫曼树是带权路径长度最小的二叉树。那么在什么情况下一棵二叉树的带权路径长度最小呢？简单地说，就是通过将权值大的外节点调整到离根节点较近的位置来得到最小路径长度。7.4.2 节将研究构造哈夫曼树的方法。

7.4.2　构造哈夫曼树

算法

哈夫曼算法，它是一种用来构建哈夫曼树的算法，哈夫曼树是哈夫曼编码的基础。

根据定义求哈夫曼树，就对给定的n个叶子节点构造出全部二叉树并求出其WPL，然后找出WPL最小的树，当n较大时，这种方法显然是不可取的。那么有没有一种算法可以直接找出哈夫曼树呢？哈夫曼提出了一种构造哈夫曼树的算法，称为哈夫曼算法。哈夫曼算法可以描述如下。

（1）根据权值$w_1,w_2,\cdots,w_n$构造n个二叉树$F=\{T_1,T_2,\cdots,T_n\}$，其中T_i是只含权值为w_i的节点。

（2）从F中选两个权值最小的二叉树T_i和T_j，构造一个根节点R，R的权$w_R=w_i+w_j$。

（3）从F中删除T_i和T_j，加入新的树R到F中。

（4）重复步骤（2）和（3），直到F中只有一棵树为止。

下面通过一个例子来说明运用哈夫曼算法来构造哈夫曼树的具体过程。如图 7-30(a)所示为一个哈夫曼树构造过程的例子。已知权值$W=\{5,6,3,9,7\}$，算法首先查找到权值最小的两个节点3和5，并生成以8为根，以节点3和5为子树的二叉树，如图 7-30(b)所示。

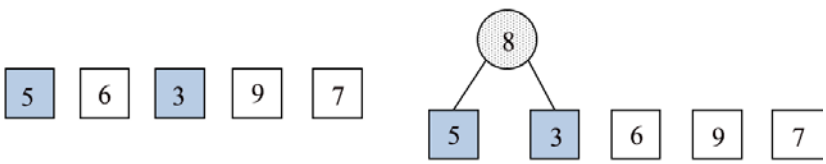


图 7-30　构造哈夫曼树（1）

由此得到$W=\{6,9,7,8\}$，继续查找具有最小权值的节点，得到6和7。重复算法步骤中的第（2）、（3）步后，$W=\{9,8,13\}$，如图 7-31(a)所示。照此规则继续下去，将节点9和8合成以17为根的二叉树，如图 7-31(b)所示。

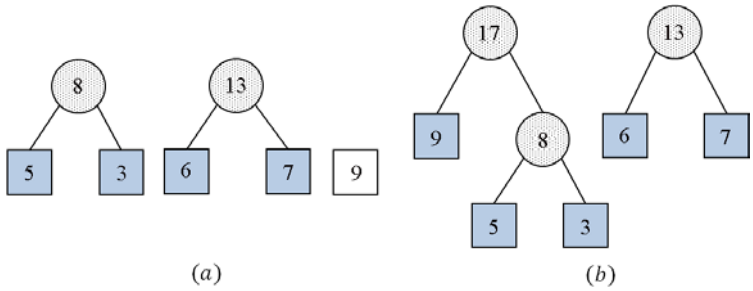


图 7-31　构造哈夫曼树（2）

最后当F中只有一棵二叉树时操作过程结束，如图 7-32 所示。最终得到的二叉树是一棵带权路径长度最小的二叉树，树中权值较大的节点位于离根节点较近的地方，而权值较小的节点则位于离根节点较远的位置。

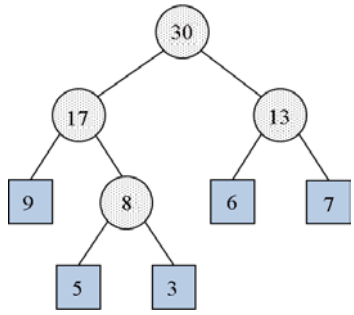


图 7-32　构造哈夫曼树（3）

7.4.3　哈夫曼编码的实现

哈夫曼树是哈夫曼编码的基础，利用哈夫曼树可以构造哈夫曼编码。哈夫曼编码的基本原理是频繁使用的数据用较短的代码代替，而较少使用的数据则用较长的代码代替，这样就可以使报文中的编码数减少到最少，从而达到压缩的目的。注意：这些编码都是用二进制数表示的，且码的长度是可变的。下面举一个例子来说明这种编码方式。

例如，给出一段文本报文：$ABCDEBBCCCDCBEDBDBDBC$。在报文中出现的字符集合为$\{A,B,C,D,E\}$，其中B出现的次数最多，为7次；其次是C，出现了6次；然后依次是D、E和A，它们出现的次数分别是5次、2次和1次。

给定5个字符编码，最少需要3位二进制编码。首先采用一种不经压缩的方法，给每个字符赋一个3位二进制编码。

A：000　B：001　C：010　D：011　E：100

则将原始报文进行编码，所得结果如下：

000 001 010 011 100 001 001 010 010 010 011 010 001 100 011 001 011 001 011 001 010

编码长度为63。

下面采用哈夫曼编码方式，将出现频率最高的*B*用最少的字符来表示，其他字符的编码类同，则上述5个字符的二进制编码如下：

B：11　　*C*：10　　*D*：01　　*E*：001　　*A*：000

上述过程所得的结果如表 7-3 所示。在此基础上将原始报文进行编码，所得的结果如下：

000 11 10 01 001 11 11 10 10 10 01 10 11 001 01 11 01 11 01 11 10

编码长度为45。采用哈夫曼编码与普通编码的压缩比为0.714。

表 7-3　哈夫曼编码

符　　号	频　　次	log(1/*p*,2)	编　　码
A	1/21(0.0476)	4.3923	000
B	7/21(0.3333)	1.5850	11
C	6/21(0.2857)	1.8074	10
D	5/21(0.2381)	2.0704	01
E	2/21(0.0952)	3.3923	001

利用构建哈夫曼树的方法可以得到报文的哈夫曼编码，具体的哈夫曼编码算法如下。

（1）统计出每个符号出现的频率，如上例中的频率统计为{1/21,7/21,6/21,5/21,2/21}。

（2）从左到右把频率按从小到大的顺序排列。

（3）每次选出最小的两个值，作为二叉树的两个叶子节点（较小的作为左叶子，较大的作为右叶子），将和作为它们的根节点，这两个叶子节点不再参与比较，新的根节点参与比较。

（4）重复上一步，直到最后得到和为1的根节点。

（5）将形成的二叉树的左节点标0，右节点标1。把从最上面的根节点到最下面的叶子节点途中遇到的0、1序列串起来，就得到了各个符号的编码。

这样就得到了一棵哈夫曼编码树，如图 7-33 所示。

当报文中高频字符的频率较高时，哈夫曼编码呈现出较高的压缩比。因此哈夫曼编码不仅应用于文本编码，同样应用在图像编码等领域中。采用哈夫曼编码的优点在于：

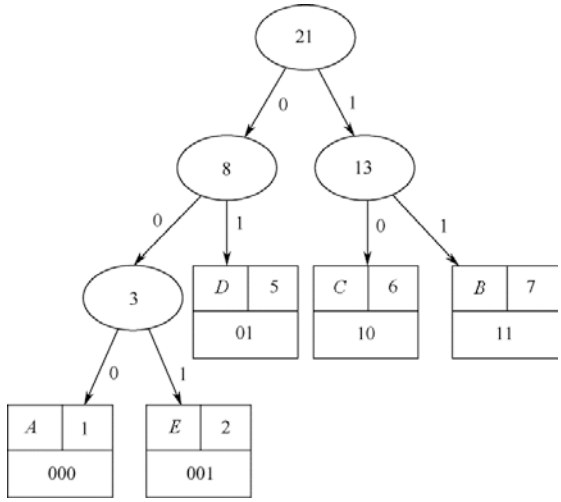


图 7-33　哈夫曼编码树

- 对于给出的报文可以得到最短编码。
- 非同一字符的任意两个字符A和B，不会出现A的编码是B的编码前缀这种情况。这是因为如果设编码字符集为$\{c_1, c_2, \cdots, c_n\}$，那么根节点到任何叶子节点$c_i$的路径都不会经过任何其他$c_k$，所以哈夫曼编码是无前缀编码，即任何一个编码都不会是另一个编码的前缀。因而两个字符之间无须分隔符。

根据上述内容易知，产生哈夫曼编码需要对原始数据扫描两遍。第 1 遍扫描是为了要统计出原始数据中每个值出现的频率，第 2 遍是建立哈夫曼树并进行编码。由于需要建立二叉树并遍历二叉树生成编码，因此数据压缩和还原速度都较慢。当然，哈夫曼编码还存在其他一些缺点。这里对哈夫曼编码的一些缺点和不足做补充说明。首先，哈夫曼编码的码字长度参差不齐，硬件实现不方便；其次，码字在存储或传输过程中，如果出现误码时，可能引起误码的连续传播；最后，对数据进行解码时必须参照哈夫曼编码表。尽管如此，由于编码方式简单且有效，因而得到了广泛的应用。

下面给出一个实现哈夫曼编码的示例程序，其中用到了 STL 容器 map，关于它的具体用法将留在本章最后进行介绍。

```
#include "stdafx.h"
#include <iostream>
#include <string>
#include <queue>
#include <map>
#include <iterator>
#include <algorithm>

using namespace std;

typedef vector<bool> HuffCode;
typedef map<char, HuffCode> HuffCodeMap;

class INode
{
```

```
public:
    const int f;
    virtual ~INode() {}
protected:
    INode(int f) : f(f) {}
};

class InternalNode : public INode
{
public:
    INode *const left;
    INode *const right;

    InternalNode(INode* c0, INode* c1) :
        INode(c0->f + c1->f), left(c0), right(c1) {}
    ~InternalNode(){
        delete left;
        delete right;
    }
};

class LeafNode : public INode
{
public:
    const char c;
    LeafNode(int f, char c) : INode(f), c(c) {}
};

struct NodeCmp
{
    bool operator()(const INode* lhs, const INode* rhs) const {
        return lhs->f > rhs->f; }
};

INode* BuildTree(map<char, int> frequs)
{
    priority_queue<INode*, vector<INode*>, NodeCmp> trees;
    map<char, int>::iterator it = frequs.begin();
    for ( ; it != frequs.end(); it++){
        trees.push(new LeafNode(it->second,  it->first));
    }

    while (trees.size() > 1){
        INode* childR = trees.top();
        trees.pop();

        INode* childL = trees.top();
        trees.pop();

        INode* parent = new InternalNode(childR, childL);
        trees.push(parent);
    }
    return trees.top();
}

void GenerateCodes(const INode* node, const HuffCode& prefix,
    HuffCodeMap& outCodes)
{
    if (const LeafNode* lf = dynamic_cast<const LeafNode*>(node)){
        outCodes[lf->c] = prefix;
```

```
    }
    else if (const InternalNode* in =
        dynamic_cast<const InternalNode*>(node)) {
        HuffCode leftPrefix = prefix;
        leftPrefix.push_back(false);
        GenerateCodes(in->left, leftPrefix, outCodes);

        HuffCode rightPrefix = prefix;
        rightPrefix.push_back(true);
        GenerateCodes(in->right, rightPrefix, outCodes);
    }
}

int _tmain(int argc, _TCHAR* argv[])
{
    map<char, int> frequs;    //创建频率表
    cout<<"请输入一行文本数据:"<<endl;
    string str;
    getline(cin, str);
    const char* ptr = str.c_str();

    while (*ptr != '\0')
    {
        if(frequs.find(*ptr)!=frequs.end())
            frequs[*ptr]=frequs[*ptr]+1;
        else
            frequs[*ptr]=1;
        *ptr++;
    }

    INode* root = BuildTree(frequs);

    HuffCodeMap codes;
    GenerateCodes(root, HuffCode(), codes);
    delete root;
    for (HuffCodeMap::const_iterator it = codes.begin(); it !
                                        = codes.end(); ++it)
    {
        cout << it->first << " ";
        copy(it->second.begin(), it->second.end(),
            ostream_iterator<bool>(cout));
        cout << endl;
    }
    system("PAUSE");
    return 0;
}
```

7.5 堆

堆是一种非常有用的数据结构类型，它可以用作执行优先级队列的存储结构，也可以用来实现堆排序。

7.5.1 堆的概念

在计算机科学中，堆是一种以完全二叉树为基础的满足堆性质的特殊数据结构，所谓

堆性质就是如果B是A的子节点，那么A中的关键码就大于等于B中的关键码。这意味着结构中的数值最大的元素总是位于树的根部，并且这样的一个堆有时也被称为最大堆。相对地，如果关键码的比较关系与此相反，即如果B是A的子节点，A中的关键码就小于等于B中的关键码，这样的一个堆被称为是最小堆，其中数值最小的元素总是位于树的根部。如图 7-34(a)所示为一个最小堆，如图 7-34(b)所示为一个最大堆。本节仅以最小堆为例进行相关说明，最大堆的情况可由最小堆推得。

小提示

本书后面介绍的堆排序算法就是以堆这种结构为基础来实现的。

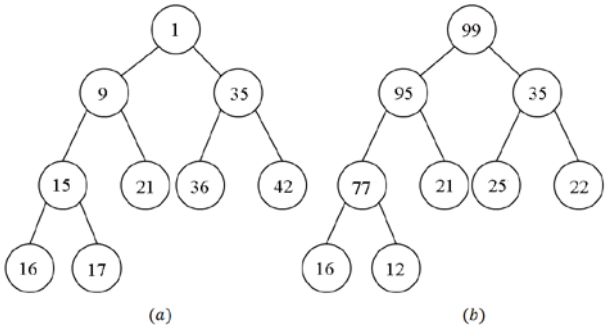


图 7-34　堆

7.5.2　堆的建立

一种建立堆的有效方法是将数据表中的元素顺序地填入一个完全二叉树中，然后通过所谓的自上而下（Top-Down）调整算法来使得该完全二叉树满足堆性质。如图 7-35 所示，我们使用了自上而下的堆调整算法使一个完全二叉树形成了一个堆结构。该算法首先找到按完全二叉树节点编号排在最后的那个分支节点，见图 7-35(a)，这样便得到了一棵子树，其节点被涂上了阴影。这时调用调整算法将以该分支节点为根的子树调整为最小堆，由于被涂为阴影的节点所构成的子树满足最小堆性质，所以该步不做调整。而后，找到按节点编号排在前面一个的分支节点（即节点 36），再调用调整算法将以该分支节点为根的子树调整为最小堆，即图 7-35(b)中的阴影部分，这时需要将节点 35 和节点 36 调换位置。此后，继续将节点编号推前一位，找到按节点编号又排在前面的一个分支节点（即节点 9），再调用调整算法将以该分支节点为根的子树调整为最小堆，处理过程如图 7-35(c)和图 7-35(d)所示。退到根节点，再调用一次调整算法，如图 7-35(e)所示。如此处理下去，直到将整棵树调整为一个堆，最终结果如图 7-35(h)所示。

可见自下而上的堆调整算法的核心思想是先将含有m个数据项的数据表按顺序存储进一棵完全二叉树中，然后再从某个分支节点i开始向下调整，如果此节点的左子节点j关键码小于右子节点$j+1$的关键码，那么就沿着节点i的左分支进行调整，其中$j=2i+1$。如果节点i的左子节点关键码大于右子节点的关键码，那么就沿着节点i的右分支进行调整。调整的方法是将该节点的关键码与其右子节点的关键码（或左子节点的关键码）进行比较，如果某个分支不满足堆性质就将该分支所在边两端的节点对调，从而把关键码较小的节点上浮。然后令$i=j$，$j=2i+1$，将比较和调整操作在局部范围内（子树内）下降一层，如果此时节点i（注意此时i不再是起始节点了，而是下降一层后的

节点）的关键码小于节点j的关键码则不做调整，算法终止。如果情况相反，就在调整后继续下降一层并做比较，直到该子树满足最小堆性质为止。最大堆的调整算法可由最小堆的调整算法类比推得，这里不再赘言。

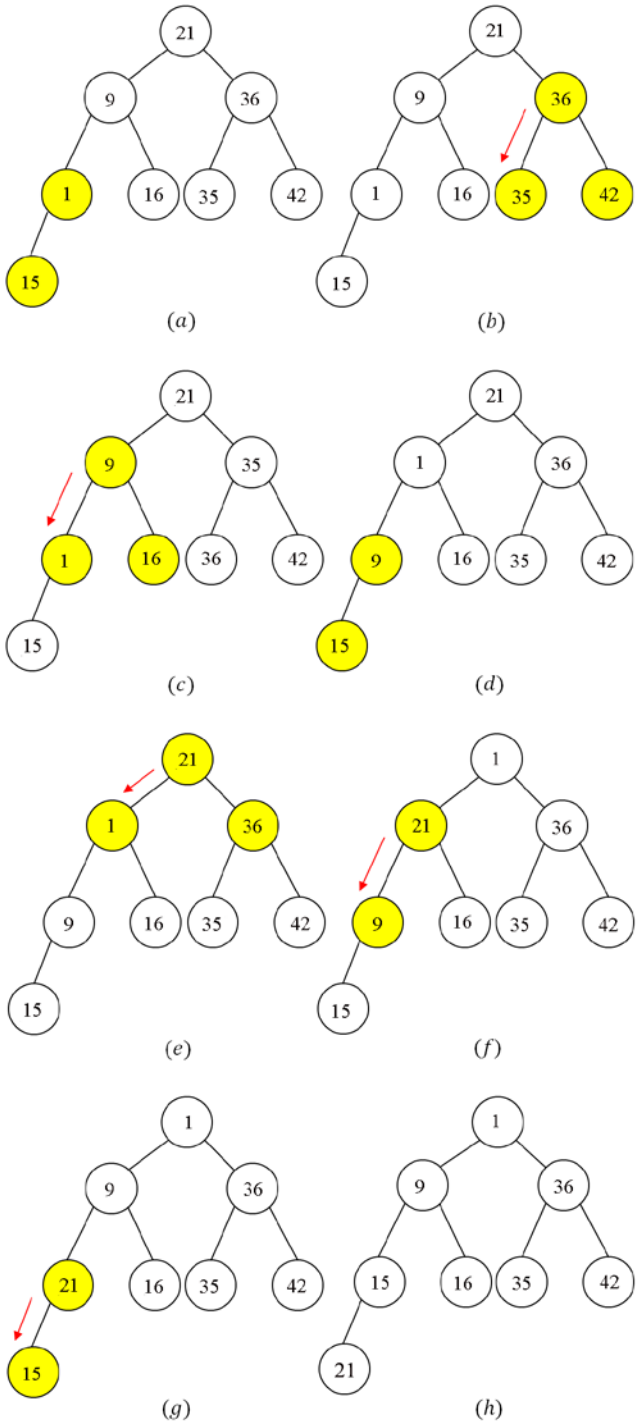


图 7-35　堆的调整算法

7.5.3　堆的操作

向最小堆中插入一个元素x的过程就是不断自下而上调整新堆的过程。每次一个被插入的新节点总是被安插在最小堆的最后面，所谓的“最后面”指的是二叉树层次遍历次序的末尾。如图 7-36(a)所示，向原最小堆中插入一个元素8。将该元素续接在最小堆的末尾后，自下而上将关键码与其父节点的关键码相比较。因为8小于15，故将节点8和节点15对调，如图 7-36(b)所示。继续此过程，在经过一次调整之后，见图 7-36(c)，节点插入结果如图 7-36(d)所示。请读者注意堆的插入算法和建堆的调整算法之间的联系和区别。

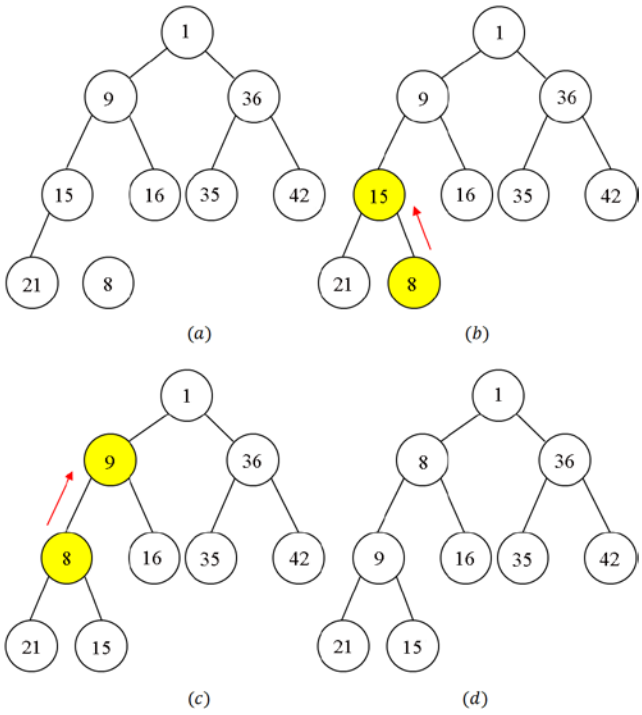


图 7-36　堆的插入操作

通常，最小堆的节点删除操作往往是指将其中具有最小关键码的数据项从堆中删除，这与优先级队列非常相似，也就是说删除操作只能在结构的一端进行。由于最小堆中关键码最小的元素总是位于堆顶，所以该删除操作实际上就是将完全二叉树的根节点删除。当然如果一棵树的根节点被删除，那么该树的结构势必被破坏。为了保持树形结构不被破坏，在删除其堆顶节点后，我们就将堆的最后一个节点（即完全二叉树层次遍历顺序下的末节点）取走用来填补缺失的堆顶元素，并将堆的实际元素个数减1。这时还要考虑使用最后一个元素取代堆顶元素后，如何维持堆的性质不被破坏。处理的方式很简单，就是使用堆的调整算法从堆顶向下进行调整，直到完全二叉树满足堆性质为止。

下面来分析堆操作的时间复杂度。因为堆的存储形式就是完全二叉树的顺序表示，所

以每删除一个堆顶元素就是删去对应完全二叉树的根节点，并用堆中最后一个节点顶替被删除节点，这些操作的时间复杂度是$O(1)$。应用堆的调整算法时，需要逐层对节点关键码进行比较，并在需要的时候进行节点对调操作。对于一棵完全二叉树的相关操作而言，堆调整算法中的循环次数最大就是树的高度，且每次循环中关键码的比较次数为2，节点对调次数为1。本章前面的知识告诉我们，具有n个节点的完全二叉树高度是$\lfloor \log_2(n+1) \rfloor$，所以堆的删除算法的时间复杂度是$O[\log_2(n)]$。而在向堆中插入一个新节点时，同样使用了一个堆的调整算法，这时堆调整算法中的循环次数将不超过树的高度，而循环内关键码比较次数和节点对调次数都是1，因此堆的插入算法的时间复杂度也是$O[\log_2(n)]$。由于建堆操作执行了$\lfloor n/2 \rfloor$次调整算法，因此其时间复杂度是$O[n\log_2(n)]$。

7.6 基于 STL 实现树结构

本书前面曾经讲过很多 STL 中的容器，它们几乎与已知的数据结构一一对应，例如可以使用 list 来实现链表，可以使用 queue 来实现队列等。那么读者可能会问 STL 中是否提供了 tree 这种容器来实现树呢？很遗憾，STL 中没有 tree 这种容器。不过 STL 中提供了其他更有用的容器，借由这些容器可以间接地实现树形结构。

7.6.1 STL 中的 vector

STL 中的 vector 类看起来有点像数组，但它提供了一个比数组更加安全且更加灵活的存储结构。之所以说 vector 类看起来像数组是因为它可以顺序地存储一系列同类型的数据。另外，由于 STL 支持泛型编程，所以 vector 中可以存储任意内置类型以及用户自定义类型的数据。

数组在使用上具有一定的不安全性，这一点本书前面曾经讨论过。数组的不安全性在于它不能进行越界检查，而 vector 在这方面则表现良好。因此 vector 较数组而言提供了更好的安全性。另一方面，数组一旦被定义那么它的容量就无法再改变，这点所带来的问题前面也曾经讨论过。而 vector 中的数据则是可以实现动态调整的，也就是说无须预先定义 vector 的容量，它能够根据实际需求动态地分配存储空间，这大大增强了编程的灵活性。

与前面所讲述过的 STL 中的容器相同，使用 vector 之前必须包含其对应的头文件。来看下面一段使用 vector 的示例代码片段。

```
#include <string>
#include <iostream>
#include <vector>

using namespace std;

int main() {

    vector<int> v1(10);
```

```
    vector<double> v2;
    vector<string> v3(10, "OK!");
    vector<string> v4(v3);

    v1[1] = 2;
    v1.at(2) = 45;

    for(int i = 0; i < 9; i++)
        v2.push_back(1.0+i);

    v2.pop_back();

    cout<<v2.back()<<endl;
    cout<<v2.front()<<endl;
    cout<<v2.at(5)<<endl;
    cout<<v1[2]<<endl;

    for(int i = 0; i < 10; i++)
        cout<<v3[i]<<" ";
    cout<<endl;
    for(int i = 0; i < 10; i++)
        cout<<v4[i]<<" ";
    cout<<endl;

    return 0;
}
```

上述代码中使用了 vector 类中的几个十分常用的方法，可能读者对它们非常感兴趣。但在介绍这些方法之前还应该先来看看 vector 类的构造函数。这段代码告诉读者 vector 类提供了许多不同的构造函数。通过它们不仅可以对其对象的大小进行初始化，甚至可以对其对象的内容进行初始化。上例中，v1 被初始化为拥有 10 个整型元素的 vector 对象，不妨测试一下这些元素的内容，易见当没有指定具体内容时它们被初始化为 0。v2 被初始化为空。v3 被初始化为包含 10 个字符串类型元素的 vector 对象，且每个元素的初始值都为“ OK! ”。v4 则被初始化为 v3 的一个副本。

vector 类提供给程序员多种不同的方式来访问其中的元素，程序员可以根据实际需求来进行选择。首先为了贴近大家所熟悉的数组，vector 类重载了操作符[]。这很容易理解，如同数组可以使用下标来访问数组中的元素一样，vector 也可以。但是 vector 类中的[]并不提供越界检查的功能，因此这可能带来某些安全隐患。不过 vector 也提供了另外一种访问其中元素的途径，即使用 at()方法，我们建议使用这种方法，因为它可以提供越界检查的功能，这对提高安全性是很有帮助的。此外，vector 类还提供了访问首元素和末元素的方法，即 front()和 back()，上面的程序中演示了这一点。

此外还需要注意的是 push_back()方法和 pop_back()方法，它们为程序设计带来了极大的便利。push_back()方法可以用于在 vector 的末尾追加一个元素。这个方法可以自动为新元素申请存储空间，这一点是数组无法比拟的。pop_back()方法的作用是删除 vector 中的最后一个元素，并自动释放存储空间。

vector 的一个非常有用的地方在于其中所包含的内容依然可以是 vector 对象，也就是说 vector 允许嵌套声明。这使得 vector 能够很容易地实现像矩阵一样的二维数据结构。

这就如同二维数组一样，但要比数组灵活许多。例如下面的声明。

```
vector<vector<int> > matrix;
```

需要注意的是，为了区别于流操作符“ >> ”，在声明嵌套式 vector 对象时，大部分编译器都要求在这两个尖括号中间添加一个空格，如上面的声明所示。

vector 容器无疑是 STL 中最常用的也最基础的一种容器。基于它可以构建一种最简单的树形结构。下面这个例子使用 vector 容器来存储每个节点的子节点指针，通过这种简单的形式，可以实现一种基于层次化的树形结构。

```
#include<vector>
#include<iostream>

using namespace std;

class TreeNode
{
    int num;
    vector<TreeNode*> * Sub;

public:

    TreeNode::TreeNode(int n)
    {
        num = n; Sub = NULL;
    }
    int getNum(){return num;}
    vector<TreeNode*> * getSub(){return Sub;}
    void setNum(int n)
    {
        this->num = n;
    }
    void setSub(vector<TreeNode*> * newSub)
    {
        this->Sub = newSub;
    }

    //其他成员方法略
};

class Tree
{
    TreeNode* root;

public:
    Tree(TreeNode* rt){root = rt;}

    void DisplayTree(TreeNode * r)
    {
        cout<<r->getNum()<<endl;

        if(r->getSub()==NULL)
        {
            cout<<"该节点叶子节点!"<<endl;
            return;
        }
        cout<<"该节点的子节点为:";
```

```
        for(int i = 0; i < (r->getSub())->size(); i++)
            cout<<(r->getSub())->at(i)->getNum()<<" ";
        cout<<endl;

        for(i = 0; i < (r->getSub())->size(); i++)
            DisplayTree((r->getSub())->at(i));
    }

    //其他成员方法略
};

int main()
{
    TreeNode treenode0(0);
    TreeNode treenode1(1);
    TreeNode treenode2(2);
    TreeNode treenode3(3);
    TreeNode treenode4(4);
    TreeNode treenode5(5);
    TreeNode treenode6(6);

    vector<TreeNode*> subOfTreeNode0, subOfTreeNode1, subOfTreeNode2;

    subOfTreeNode0.push_back(&treenode1);
    subOfTreeNode0.push_back(&treenode2);

    subOfTreeNode1.push_back(&treenode3);

    subOfTreeNode2.push_back(&treenode4);
    subOfTreeNode2.push_back(&treenode5);
    subOfTreeNode2.push_back(&treenode6);

    treenode0.setSub(&subOfTreeNode0);
    treenode1.setSub(&subOfTreeNode1);
    treenode2.setSub(&subOfTreeNode2);

    Tree tree(&treenode0);
    tree.DisplayTree(&treenode0);

    return 0;
}
```

上面这段示例代码利用 vector 容器有效地组织了树结构中的节点。作为演示之用，这里并没有完整地将节点类和树类全部实现，有兴趣的读者不妨自己填充并改进这两个类。主函数中利用了 vector 容器的 push_back 方法来将每个节点的子节点集合有效地组织在一起进而形成一棵树。当然这些地方最好是改成类的方法或者在构造函数中完成比较妥当，这里给出的仅仅是一种朴素的实现方式。此外，还请读者留意树类中的 DisplayTree()方法，它实现的功能是树形结构的层次遍历。

尽管用上述方法来实现树十分直观且很容易理解，但这并不是一个非常好的选择，因为基于这种设计来实现树的某些算法可能非常烦琐，当然稍加改进也可以克服一些问题。本章前面曾经介绍过多种树的存储表示方式，其实它们很多都可以利用 vector 来实现。这很容易理解，因为树的很多实现方式中都用到了线性表，而 vector 就是一种线性表。前面关于树的存储表示已经讲过很多，只要读者能够正确理解它们，具体实

现应该就不是一个大问题了。

7.6.2 STL 中的 map

STL 中的 map 容器是一种非常特殊的存储结构，map 中存储的每个数据都是以键值-数值对的形式存在的。map 容器被认为是一种关联性结构，因为它构建了一种从键值到数值之间的映射关系。

声明一个 map 型的对象需要同时给定键值和数值数据类型。键值由第 1 个模板参数指定，而数值则由第 2 个模板参数指定。例如，下面这段代码分别声明了两个 map 对象。第 1 个 map 对象提供的是从 string 类型到 string 类型的映射关系，而第 2 个则提供了从 string 类型到整型之间的映射关系。

```
map<string, string> map0;
map<string, int> map1;
```

因为 map 容器中的每个数据都是一个键值-数值对，所以向 map 容器中插入一个元素必须同时提供键值和数值两个数据。map 类提供了 insert 方法来用于向其中插入数据。来看下面的示例代码片段。

```
map0.insert(pair<string, string>("China", "an eastern country"));
map0.insert(pair<string, string>("Denmark", "a western country"));
```

上述代码向对象 map0 中插入了两个元素。第 1 个元素的键值是“ China ”，由它映射的数值是“ an eastern country ”。第 2 个元素的键值是“ Denmark ”，由它映射的数值是“ a western country ”。需要注意的是被插入的元素在类型上应当与声明使用的数据类型相一致。

使用 insert 方法向 map 容器中插入数据是一个非常有效的选择，但是这个方法相对而言实在非常烦琐且麻烦。map 类提供了另外一种可选的向其对象中插入元素的方法，即使用重载操作符“ [] ”。下述这段代码实现了与前面代码同样的功能。

```
map0["China"] = " an eastern country";
map0["Denmark"] = " a western country";
```

重载的操作符“ [] ”同时也提供了一种根据键值来访问数值的方法，无疑这是一个十分简便的访问方法。例如下面的语句可以输出前面插入元素的数值。

```
cout << map0["China"] << endl;
cout << map0["Denmark"] << endl;
```

此外，map 类还提供了 size、find 和 count 等成员函数。下面这段示例代码演示了这些常用函数的使用方法。注意在使用 map 之前必须包含相应的头文件。

```
#include<map>
#include<string>
#include<iostream>

using namespace std;

int main()
{
```

```
    map<string, string> writer;

    writer["Shakespeare"] = "English writer,
        his works include 'Hamlet', 'Othello', and 'King Lear', etc.";
    writer["Tagore"] = "India writer,
        his works include 'Gitanjali', etc.";
    writer["Tolstoy"] = "Russian writer,
        his works include 'War and Peace', and 'Anna Karenina', etc.";
    writer["Andersen"] = "Danish writer,
        his works include 'The Ugly Duckling', etc.";
    writer["Dumas"] = "French writer,
        his works include 'The Count of Monte Cristo',
        and 'The Three Musketeers', etc.";

    cout<<writer.size()<<endl<<endl;

    int i = 1;

    map<string, string>::iterator it = writer.begin();
    for ( ; it != writer.end(); it++)
    {
        cout << i << " : " <<endl;
        cout << it->first << ": " << it->second << endl;
        i++;
    }

    cout<<endl;
    cout<<(writer.find("Tagore"))->second<<endl<<endl;

    writer.erase(writer.find("Tagore"));
    cout<<writer.size()<<endl<<endl;

    writer.clear();
    cout<<writer.size()<<endl;

    return 0;
}
```

观察运行结果，读者应该不难发现遍历存储在 map 对象中的元素的一个途径是使用迭代器 iterator。特别注意，成员函数 begin()和 end()分别获得容器中的首元素和末元素的迭代器。通过它们来控制访问的界限就能够很容易地对整个 map 容器进行遍历。由于 map 容器内部的每个元素都是以键值-数值对的形式存在的，因此，一个指向 map 容器中某元素的迭代器实际上指向的就是一个 pair<key, value> 类型的对象。为了通过迭代器访问一个元素的键值，我们必须废弃原迭代器而使用键值-数值对对象的第 1 个数据成员。同理，该对象的第 2 个数据成员提供的是对于相应数值的访问。

此外，通过程序运行结果可以看出 map 容器根据对象的键值将其进行了排序。因此我们向 map 容器中插入对象的顺序与输入对象的顺序无关。

下面来对上述代码中所使用的成员函数的功能和用法进行说明。

- size()

该函数返回被控序列的长度。

- erase(iterator it)

该函数将从被控序列中删除由迭代器 it 所指定的元素。此外，该函数还有另外一个重载形式，即 erase(iterator first, iterator last)，它将删除区间(first, last)中的所有元素。这两种重载形式都返回一个迭代器，它指向紧接着被删除元素的下一个元素，如果没有这样的元素则指向 end()。

- clear()

该函数调用 erase(begin(), end())，即将整个容器清空。

- find(const Key & key)

该函数返回一个迭代器，它指向被控序列中键值与 key 相等的元素。如果不存在这样的元素，函数将返回 end()。

在后续章节中，读者会发现 map 不仅可以用来实现树，甚至还可以用来实现图。

最后提醒读者注意存储在 map 对象中的键值必须是唯一的，且每个键值只能映射一个数值。此外，STL 中还提供了一种 multimap 容器，与 map 不同，multimap 允许键值映射多个数值，限于篇幅，本书不会对 multimap 进行介绍，有兴趣的读者可以参阅其他资料以了解更多。

本章参考文献

[1] David A. Huffman. A Method for the Construction of Minimum-Redundancy Codes. Proceedings of the IRE, Vol. 40, No. 9: 1098-1101, 1952.

[2] 左飞. 数字图像处理 :技术详解与 Visual C++实践. 北京 :电子工业出版社, 2014.

[3] 王红梅，胡明，王涛．数据结构（C++版）．北京：清华大学出版社，2011.

第 8 章

图——始于哥尼斯堡的七桥问题

作为离散数学的一个重要分支，图论已有两百多年的历史。由于其广泛的应用背景，近半个世纪以来，越来越多的科研工作者投入到了该领域的研究中。特别是伴随着计算机的出现，有关图的理论有了更加迅速的发展和扩充。图论现在已经成为研究系统工程、管理科学与工程、计算机科学、通信与网络理论、自动控制、运筹学以及社会科学等诸多学科的一种重要数学工具。图论的应用之所以如此广泛，是由于它能够为解决许多具体问题提供非常理想的离散数学模型。前面已经讲过的树结构就是图论中一类重要的研究对象。本章介绍另一种非线性离散数据结构——图。需要说明的是，我们并不打算将重点放在图结构及其接口的实现上，而是打算从图的某些操作需求出发，引出一些重要的算法，所以读者也应该将关注的重点放在对这些算法的理解上。

8.1 图的基本概念

本节介绍有关图的一些概念，而这里的图并非生活中所说的图画或地图，而是用一些点和线来表示事物间关系的某种抽象模型。与前面介绍的树不同，图不再是体现分层特点的有根结构，它体现的是一种多对多的关系模型。图的性质作为图论研究的一个主要内容，具有重要的理论及应用价值，比如大规模集成电路的分析与设计、印刷电路板的设计与布线、传递网络和通信网络稳定性与可靠性研究等，这些问题都可以利用图来解决。

8.1.1 图的定义

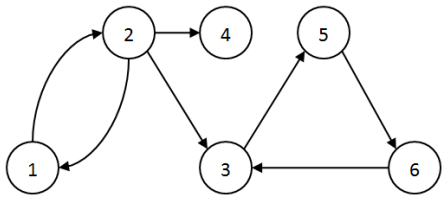
图论所研究的图（Graph）是一种抽象的模型，它是事物及其相互之间联系的一种抽象表示，不是日常的地理位置，本质上就是集合与其上关系的图形表示。

图G由两个集合$V(G)$和$E(G)$及它们二者之间的联系组成，记为$G=(V,E)$，其中V是顶点的非空有限集，E是边的有限集合，边是顶点的无序对或有序对。元素V称为顶点，元素E称为边，E中的每一条边连接V中两个不同的顶点。

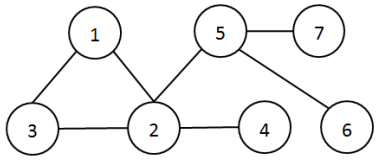
根据图中边的性质，可以把图分为有向图和无向图两种。所谓有向与无向是针对图中的顶点对是否有序而言的。如果图中的顶点对是有序的，则将有序的顶点对用$\langle v,w\rangle$表示，其中v、w为顶点，v为弧尾，w为弧头。若图中的顶点对是无序的，则将无序的顶点对用(v,w)表示。如果图中所有的边都是有向的，那么该图称为有向图；相反，

如果图中所有的边都是无向边，那么该图称为无向图；对于部分边是有向，而另一部分边是无向的图，则称为混合图。

使用顶点对$\langle i,j\rangle$或(i,j)可以表示出图中的任意一条边，其中i和j是边E所连接的两个顶点。图 8-1 是两个图的例子，其中 8-1(a)是一个有向图，图 8-1(b)是一个无向图。可见，在图 8-1(a)中，$V(G_1)=\{1,2,3,4,5,6\}$，$(G_1)=\{\langle 1,2\rangle,\langle 2,1\rangle,E\langle 2,3\rangle,\langle 2,4\rangle,\langle 3,5\rangle,\langle 5,6\rangle,\langle 6,3\rangle\}$；在图 8-1 (b) 中，$V(G_2)=\{1,2,3,4,5,6,7\}$，$E(G_2)=\{(1,2),(1,3),(2,3),(2,4),(2,5),(5,6),(5,7)\}$。



(a) 有向图



(b) 无向图

图 8-1　有向图和无向图（无向加权图的例子）

由定义可知，一个图中不可能包括同一条边的多个副本，因此，在无向图中的任意两个顶点之间，最多只能有一条边。在有向图中的任意两个顶点之间，最多只能有一条边从顶点i到顶点j或从j到i。如果一个图中包含自连边，即(i,i)类型的边，则称这种边为环，本书不讨论这种带环图。通常把无向图简称为图，有向图仍称为有向图。

8.1.2　图的术语

本节介绍一些和图有关的术语，它们将帮助读者更好地认识和利用图这种结构。

- 权：与图的边或弧相关的数。

在一些图和有向图的应用中，会为每条边赋予一个权或耗费，它们可以用来表示从一个顶点到另一个顶点的距离、所需的时间或花费的代价等。

- 网络：带权的图。

如果一个图中的边是带权边，那么这个图就称为加权图或网络。实际上，这里定义的所有图都可以看作网络的一种特殊情况，即一个无向（有向）图可以被看作是一个所有边具有相同权的无向或有向网络。加权图的结构与现实生活紧密联系，因此它通常具有更加广泛的应用。图 8-2 给出了一个无向加权图的例子。它表示一张区域地图，图中的顶点为城市，无向边代表两个城市间的连通关系，边上的权代表公路造价。在

分析了这张图后发现，任一对城市都是连通的。现在要求用公路把所有城市联系起来，如何设计才可使得工程的总造价最低？这是典型的无向图最小生成树问题，后续的内容会对此类问题的解法做进一步讲解。

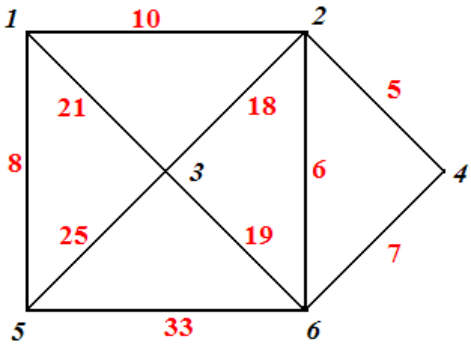
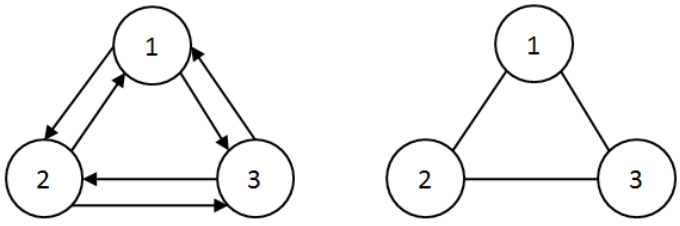


图 8-2　城市连接图（无向加权图的例子）

- 有向完备图与无向完备图

如果在一个有n个顶点的有向图中，图的边数是$n(n-1)$，那么这个图就是一个有向完备图，也称作有向完全图，如图 8-3(a)所示。

如果在一个有n个顶点的无向图中，图的边数是$n(n-1)/2$，那么这个图就是一个无向完备图，也称作无向完全图，如图 8-3(b)所示。简单地说，无向完备图就是任意顶点都与其他所有顶点相连的无向图。



(a) 有向完全图　　(b) 无向完全图

图 8-3　完全图

- 子图：如果图$G(V,E)$和图$G'(V',E')$，满足$V' \subseteq V$，$E' \subseteq E$，则称G'为G的子图。

定义说明如果图G'被完全包含在图G中，那么G'就是G的子图，显然任何图总是它自身的一个子图。图 8-4 给出了一个子图的例子，其中图 8-4(b)是图 8-4(a)的子图。对于图$G(V,E)$和图$G'(V',E')$，如果G'为G的子图，且$G' \neq G$，则称G'是G的真子图。当$V'=V$，$E' \subseteq E$时，则称G'是G的生成子图。

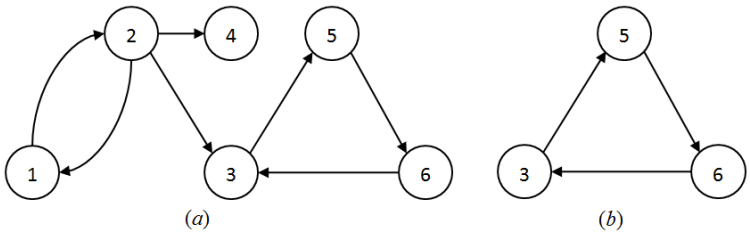


图 8-4　子图

- 顶点的度

在无向图中，顶点的度为与每个顶点相连的边数，记为$\deg(v)$。在有向图中，顶点的度分成入度与出度。对于任意顶点v，以v为起始点的边的数目称为顶点v的出度，记为$\deg^+(v)$；而以v为终止点的边的数目相应地就是顶点v的入度，记为$\deg^-(v)$。在图 8-1(a)中，有向图顶点 2 入度为 1，出度为 3；顶点 4 入度为 1，出度为 0。在图 8-1(b)中，无向图顶点 5 的度为 3，顶点 2 的度为 4。

各个顶点的度都相等的无向图也称为正则图，并记作k-正则图，其中k表示各顶点的度。图 8-5 是一个 3-正则图，它也被称为彼得森图。

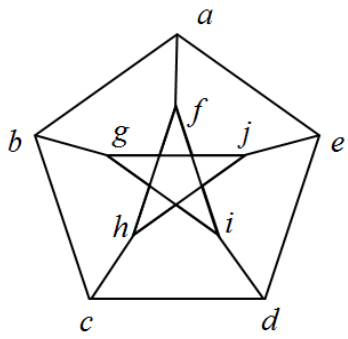


图 8-5　彼得森图

- 路径

在图$G=(V,E)$中，路径是从顶点v_0出发，途经一条点边交替序列到达终点v_n的通路。因此路径可以用一个点边交替序列来表示，如$(v_0e_1v_1e_2v_2\cdots e_nv_n)$。如果路径中没有重复边，那么这条路径叫作简单路径，如果路径中没有重复顶点那么这条路径就叫作基本路径。显然，基本路径也一定是简单路径。如图 8-6 所示，路径$(v_5e_5v_1e_1v_2e_6v_5e_4v_4e_3v_3e_2v_2)$是一条简单路径，路径$(v_1e_1v_2e_2v_3e_3v_4e_4v_5)$是一条基本路径。

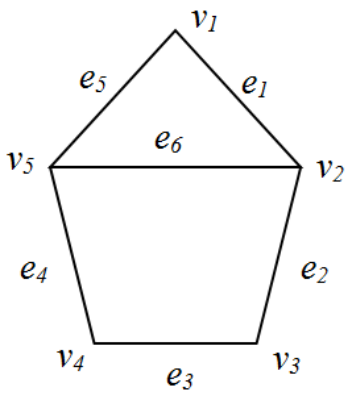


图 8-6　简单路径与基本路径

通常将沿路径边的数目或沿路径各边权值之和称为路径长度。特别地，对于一条路径，如果它的起始顶点与终点相同，那么这条路径称为一条回路。没有重复边的回路叫作简单回路，除起点和终点外，没有其他重复顶点的回路叫作基本回路。

如果图中任意两顶点之间的边都不大于一条，那么也可以用顶点序列来作为路径的简便表示。例如图 8-1(a)中，从顶点 1 到顶点 6 的路径是 1→2→3→5→6，同时它也是一条简单路径，路径长度为 4。此外从点 1 到点 6 的路径还可以是 1→2→3→5→6→3→5→6，但它却不是一条简单路径。由此也不难看出在某个图中，对于从一个顶点到另外一个顶点的路径可能有多条。在图 8-1(b)中，路径 1→2→3→1 是一条回路，它同时也是一条简单回路。

- 连通与连通图

图的连通性是实际应用中非常重要的一种性质。连通性的定义是这样的，若从顶点v到顶点w有一条路径，则说v和w是连通的。如果无向图中任意两个顶点都是连通的，那么就说这个图是一个连通图。若一个无向图不是一个连通图，那么其中非连通图的每一个连通部分都称作是一个连通分量，即连通分量是无向图中的极大连通子图。

求职面经点拨

请试回答附录题库中的第 27 题。

对于有向图$G=(V,E)$，如果对每一对$V_i,V_j \in V$，且$V_i \neq V_j$，满足从V_i到V_j和从V_j到V_i都存在路径，则称这个有向图为一个强连通图。如图 8-4(b)所示就是一个强连通图。特别地，有向图中的极大强连通子图也被称为强连通分量。

- 稀疏图与稠密图

稀疏图和稠密图是两个相对的定义。通常认为边较少的图就是一个稀疏图；相对地，接近完全图的图就是稠密图。注意这里给出的定义仅仅是一个形式化的描述，没有一个十分完备的标准来界定到底多少才算稀疏。之所以提出稀疏图和稠密图的概念，是因为稀疏图可以使用稀疏矩阵来表示，进而节省存储空间。

- 多重图与线图

在介绍多重图的概念之前首先给出平行边的定义。对于无向图，如果两顶点之间的边

多于一条，则称这几条边是平行边。在有向图中，如果从同起始点到同终止点的边多于一条，那么这几条边是平行边。含有平行边的图称为多重图。在多重图中，任意两顶点之间的边的数目称为这两点间的重数。不含有平行边的图称为线图。特别地，不含有回路的线图称为简单图。本章所讨论的图大多数是线图。

前面介绍了许多图中的术语，下面来看一个被称为“握手定理”的结论。在这个定理的描述和证明过程中用到了许多前面提过的术语，读者可以借此深化一下理解。

握手定理：在无向图中，度数为奇数的顶点必然有偶数个。

证明：在无向图$G=(V,E)$里，设V_1和V_2分别是偶数度顶点和奇数度顶点的集合，那么如果度数为偶数的顶点有n_1个，度数为奇数的顶点有n_2个，则

$$n_1=\sum_{v\in V_1}\deg(v),\qquad n_2=\sum_{v\in V_2}\deg(v)$$

在一个无向图中，因为每条边都提供了两个度，而所有顶点的度数之和都是由m条边提供的，因此也就得到下式必然成立。

$$2m=\sum_{v\in(V_1+)}\deg(v)$$

其中：

$$\sum_{v\in(V_1+V_2)}\deg(v)=\sum_{v\in V_1}\deg(v)+\sum_{v\in V_2}\deg(v)=n_1+n_2$$

由于$2m$是一个偶数，所以可知n_1+n_2必然也是一个偶数。因为度数为偶数的各个顶点的度数和是偶数，即对于$v\in V_1$来说n_1是偶数。而n_1+n_2也是偶数，由基本的数学知识就可得到n_2也是偶数。n_2是若干个奇数的和，又知奇数个奇数的和是奇数，偶数个奇数的和是偶数，所以n_2必然是偶数个奇数的和。结论得证，即无向图中有偶数个奇数度顶点。

通过这个定理可以得到一个简单的结论：在任何聚会中，与奇数个人握过手的人必定有偶数个。用点来表示聚会中的人，若两个人有握手，则用一条边把表示这两个人的点连接起来。与奇数个人握过手的那个人，在图上表示为他的度为奇数。由握手定理，则可知这样的点应该有偶数个，所以与奇数个人握过手的人必定有偶数个。这也就是“握手定理”得名的来源。

8.1.3 图的运算

可以在图上定义一些运算。最基本的运算包括删除图上的边、删除图上的点、在图上增加一条边、在图上增加一个点等。但本节将讨论一些其他的操作。我们已经知道，图是由边集和点集以及它们之间的关联映射组成的结构，因此图可进行与集合运算相类似的运算。图的常见运算有并、交、差、环和等。对于图$G_1=\langle V_1,E_1\rangle$，$G_2=\langle V_2,E_2\rangle$，有下列运算。

- 并运算

G_1与G_2的并运算，记为$G_1 \cup G_2 = G_3 = \langle V_3, E_3 \rangle$，其中，$V_3 = V_1 \cup V_2$，$E_3 = E_1 \cup E_2$。

- 交运算

G_1与G_2的交运算，记为$G_1 \cap G_2 = G_3 = \langle V_3, E_3 \rangle$，其中，$V_3 = V_1 \cap V_2$，$E_3 = E_1 \cap E_2$。

- 差运算

G_1与G_2的差运算，记为$G_1 - G_2 = G_3 = \langle V_3, E_3 \rangle$，其中，$E_3 = E_1 - E_2$，$V_3 = (V_1 - V_2) \cup \{E_3$中边所关联的顶点$\}$。

- 环和运算

G_1与G_2的环和运算，记为$G_1 \oplus G_2 = G_3 = \langle V_3, E_3 \rangle$，其中，$G_3 = (G_1 \cup G_2) - (G_1 \cap G_2)$。

下面举例说明上述运算。假设有图G_1和G_2如图 8-7 所示。

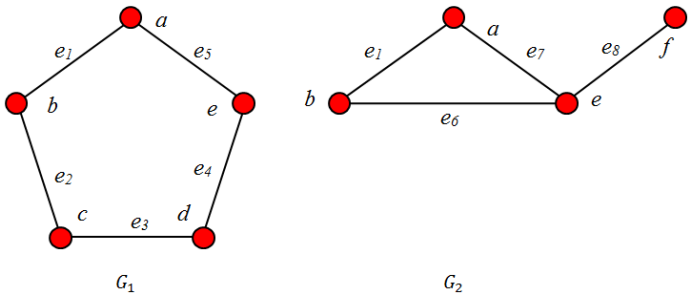


图 8-7　图G_1和G_2

图 8-8 演示了G_1与G_2的并、交、差运算。环和运算留给读者思考。

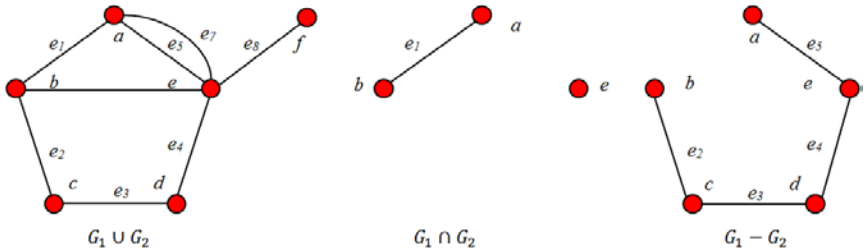


图 8-8　图的运算

图的定义与运算决定了图应有的数据结构，图的抽象数据类型应该充分考虑图的结构与运算，8.1.4 节将介绍图的抽象数据类型。

8.1.4　图的抽象数据类型

图的操作纷繁而复杂，因此图的抽象数据类型也具有非常多的操作接口，为了实现不同算法而设计的抽象数据类型接口也有可能存在差异。这里给出的抽象数据类型只提

供了较为基础的操作，读者可以根据实际需求为该类型添加新的操作。下面给出图的抽象数据类型。

```
class Graph()
{
    vector<Vertex> VertexSet; //点集，这里使用 vector 作为存储结构
    vector<Edge> EdgeSet;     //边集，这里使用 vector 作为存储结构

public:
    Graph();                                        //默认构造函数，创建一个空图
    Graph(vector<Vertex> vertexes, vector<Edge> edges); //含参数的构造函数
    ~Graph();                                       //析构函数

    bool IsEmpty();       //判断图是否为空，若是则返回 true，否则返回 false

    void InsertVertex(const Vertex& vertex);    //在图中插入新的顶点
    void InsertEdge(const Edge& edge);          //在图中插入新的边

    void DeleteVertex(Vertex& vertex); //删去图中的一个顶点，以及与该顶点相关联
                                       //的边
    void DeleteEdge(Edge& edge);       //删去图中的一条边

    Vertex * getVertex(const string& nv);//获取图中某一顶点，若无此点则返回 NULL
    Edge * getEdge(const string& ne);  //获取图中某一条边，若无此边则返回 NULL

    int getEdgeValue(Edge& edge);      //返回一条边的权值
    Vertex * getEdgeSrc(Edge& edge);   //返回一条边的起始点
    Vertex * getEdgeDst (Edge& edge);  //返回一条边的终止点
    //获得顶点 vertex 的首个邻接顶点，若无则返回 NULL
    Vertex * getFirstAdjVex(Vertex& vertex);
    //获得顶点 vertex1 的某个邻接顶点 vertex2 的下一个顶点，若无则返回 NULL
    Vertex * getNextAdjVex(Vertex& vertex1, Vertex& vertex2);
    //从顶点 vertex 开始对图进行深度遍历
    void DFSTraverse(Vertex& vertex);
    //从顶点 vertex 开始对图进行广度遍历
    void BFSTraverse(Vertex& vertex);
    //获得从顶点 vertex1 到顶点 vertex2 的一条最短路径
    vector<Edge*> getShortPath(Vertex& vertex1, Vertex& vertex2);
}
```

下面定义图的顶点结构。

```
class Vertex
{
    string name_vertex;

    Vertex(const string * n)
    :name_vertex (n) {}
}
```

在顶点结构的基础上可以定义图的边结构。

```
class Edge
{
    string name_edge;
    Vertex * src;
```

```
    Vertex * dst;
    int cost;

    Edge(const string * n, Vertex * s = NULL, Vertex * d = NULL, int c = 0)
    : name_edge(n), src(s), dst(d), cost(c) {}
}
```

以上即定义了图的抽象数据类型，其中部分接口的实现算法会在后面的内容中讨论，但限于篇幅，本章不再给出该类型接口的完整实现，读者可以自己尝试填充它们。此外，读者也可以尝试添加其他操作接口，并尝试着实现图的交、并、差等运算。这将有助于读者深化理解图这种数据结构。

8.2　图的存储与表示

本节将讨论图的存储和表示等方面的内容。矩阵是研究图性质的重要数学工具。使用矩阵可以很方便地存储图结构，矩阵还为研究图的路径和回路等问题提供了便利，本节将重点介绍图的邻接矩阵表示法。常用的表示图的方法还有邻接表，本节也将对其进行介绍。

8.2.1　图的邻接矩阵表示

一个具有n个顶点的图$G=(V,E)$，其中$V=\{v_0,v_1,\cdots,v_{n-1}\}$，可用如下形式的一个$n\times n$矩阵$\boldsymbol{A}$描述，对于$\boldsymbol{A}$中的每一个元素$a_{ij}$，满足

$$a_{ij}=\begin{cases}1, & (v_i,v_j)\in E\text{或}\langle v_i,v_j\rangle\in E\\ 0, & (v_i,v_j)\notin E\text{或}\langle v_i,v_j\rangle\notin E\end{cases}$$

由于矩阵$\boldsymbol{A}$中的元素a_{ij}表示了顶点v_i和顶点v_j之间边的关系，即$\boldsymbol{A}$中的元素a_{ij}表示了顶点v_i和顶点v_j的邻接关系，所以矩阵$\boldsymbol{A}$称作邻接矩阵（Adjacency Matrix）。图 8-9(a)是一个无向图邻接矩阵的例子，图 8-9(b)是一个有向图邻接矩阵的例子。

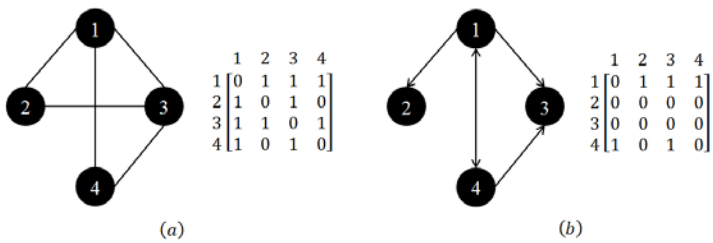


图 8-9　邻接矩阵

易见，无向图的邻接矩阵一定是一个对称矩阵，有向图的邻接矩阵则不一定，而且通常情况下都是非对称的。

下面将邻接矩阵推广到多重图和带权图。

在多重图中，邻接矩阵$\boldsymbol{A}$中的a_{ij}表示的是两顶点间的重数。图 8-10 是一个多重图的邻

接矩阵的例子。

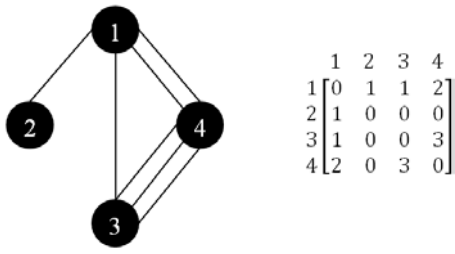


图 8-10　多重图的邻接矩阵

对于无向带权图，邻接矩阵$\boldsymbol{A}$中的a_{ij}表示的是两顶点间边的权值。特别规定当两顶点之间没有边的时候，$a_{ij}=\infty$，处于对角线上的$a_{ij}=0$。图 8-11 是一个无向带权图的邻接矩阵的例子。

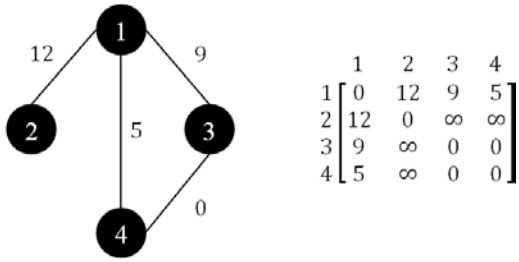


图 8-11　无向带权图的邻接矩阵

对于有向带权图，邻接矩阵$\boldsymbol{A}$的定义如下：

$$a_{ij}=\begin{cases}W\langle i,j\rangle, & i\neq j\text{且}\langle i,j\rangle\in E\\ 0, & i=j\\ \infty, & i\neq j\text{且}\langle i,j\rangle\notin E\end{cases}$$

其中，$W\langle i,j\rangle$表示边$\langle i,j\rangle$的权值。

使用邻接矩阵对图进行研究非常方便，而且通过一些简单的矩阵运算往往可以得到一些有趣的结论。下面就给出一个计算有向图出度与入度的结论。假设有向图$G=(V,E)$及其邻接矩阵$\boldsymbol{A}$如图 8-12 所示。

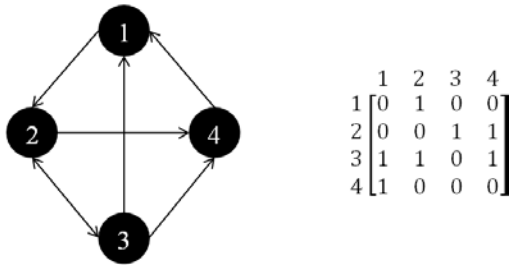


图 8-12　有向图 G 及其邻接矩阵 A

由线性代数知识，可以计算$\boldsymbol{A}\boldsymbol{A}^{\mathrm{T}}$和$\boldsymbol{A}^{\mathrm{T}}\boldsymbol{A}$如下：

$$\boldsymbol{AA^{\mathrm{T}}}=\begin{bmatrix}1&0&1&0\\0&2&1&0\\1&1&3&1\\0&0&1&1\end{bmatrix},\qquad \boldsymbol{A^{\mathrm{T}}A}=\begin{bmatrix}2&1&0&1\\1&2&0&1\\0&0&1&1\\1&1&1&2\end{bmatrix}$$

那么$\boldsymbol{AA^{\mathrm{T}}}$和$\boldsymbol{A^{\mathrm{T}}A}$具有什么意义呢？观察后不难发现，对于$\boldsymbol{AA^{\mathrm{T}}}$，对角线上的元素$b_{ij}$，其中$i=j$，表示的就是顶点$v_i$的出度。相应地，对于$\boldsymbol{A^{\mathrm{T}}A}$，对角线上的元素$b_{ij}$，其中$i=j$，表示的就是顶点$v_i$的入度。这个结论很容易证明，读者可以自己尝试推导，这里不再赘述。

下面给出使用邻接矩阵表示图的存储结构 GraphMatrix 的定义。

```
typedef int Vertex;
typedef int AdjMatrix;
class GraphMatrix
{
    int n;                  //图中顶点的数目
    Vertex * vexes;         //图中的顶点被存在一个数组里
    AdjMatrix * arcs[];     //用矩阵存储图中的边
}
```

8.2.2　图的邻接表表示

尽管使用邻接矩阵来存储图结构有很多应用上的便利之处，但是显然这种存储方式的时空开销相对较高。特别是当图的规模很大时，也就是当图中的顶点较多时，这种方式的时空开销会明显激增。可否考虑一种更加经济的存储方式呢？在本章 8.1 节中曾经给出了稀疏图的概念。如果一个图是稀疏图，即图中的边数少于顶点个数，这时其邻接矩阵中就会出现大量的 0 元素，图的邻接矩阵就变成了一个稀疏矩阵。从这个角度来考虑优化存储结构，不妨设法将稀疏矩阵中的冗余零元素进行优化和压缩，从而节省存储空间。

通过以上分析，可以将原邻接矩阵改进成一种邻接表的形式。这个用来存储图结构的邻接表由两部分组成。

首先是存储图中顶点信息的顺序表，该表又称为顶点表。顶点表存放顶点本身的数据信息，表中每个项目对应于图中的一个顶点，因此，具有n个顶点的图的邻接表中，顶点表必然有且仅有n项。顶点表的每项包括两个字段，即顶点字段和指针字段。其中，顶点字段存放顶点V_i的信息，指针字段存放与V_i相关联的边表中的第 1 个边顶点的位置。

邻接表的另一部分是与每个顶点相关联的链式存储的边信息表。边信息表把同一个顶点出发的边链接在一个单链表中，因此又称为边链表，可想而知，在具有n个节点的图的邻接表中应当具有n条边链表。

每条边链表中的表头信息都分别与顶点表中的项目一一对应，它们存储的是图中的一个顶点V_i以及指向下一个链节点的指针，随后的链节点中存放的就是与V_i相关联的顶点V_j的下标和指向同一链表中下一个链节点的指针。边链表中的表头信息与其随后任

意一个链节点中储存的顶点V_j构成的序偶即表示图中的一条边(V_i, V_j)。对于带权图，权字段存放有边顶点所表示的边的权值（如果不是带权图，则应该省略该字段），该字段位于顶点字段的后面，指针字段指向边表的下一个边节点。图 8-13 给出了一个边链表的例子。

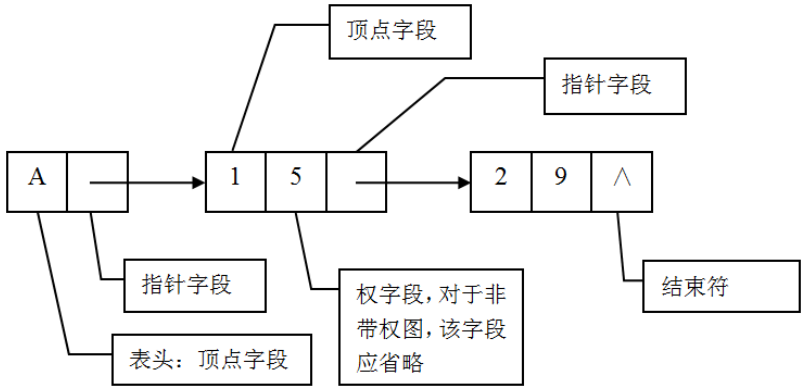


图 8-13　边链表

前面的描述似乎都是在针对无向图而言的，那么有向图如何使用邻接表来表示呢？无向图的边没有方向，使用一组边表就可以将图中的边表示完全。但有向图的边是有方向的，于是仅仅使用一组边表是不够的。有向图的边表通常分为两组，一组称作出边表，一组称作入边表。出边表用来记录从任意顶点发出的边的信息，而入边表则用来记录以某顶点为终止点的边的信息。

如图 8-14 和图 8-15 所示为图结构及其邻接表的例子，其中图 8-14 是一个无向图及其对应的邻接表，邻接表的最左侧是顶点表，顶点表中的每个表项都引出了一个边链表，边链表中的顶点信息段记录的是顶点在顶点表中的下标。

图 8-15 是一个有向图及其邻接表的例子。分析可知，在图 8-15 中，(a)表示的是入边表，(b)表示的是出边表。

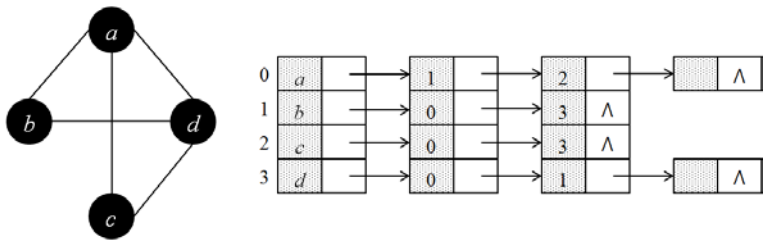


图 8-14　无向图的邻接表

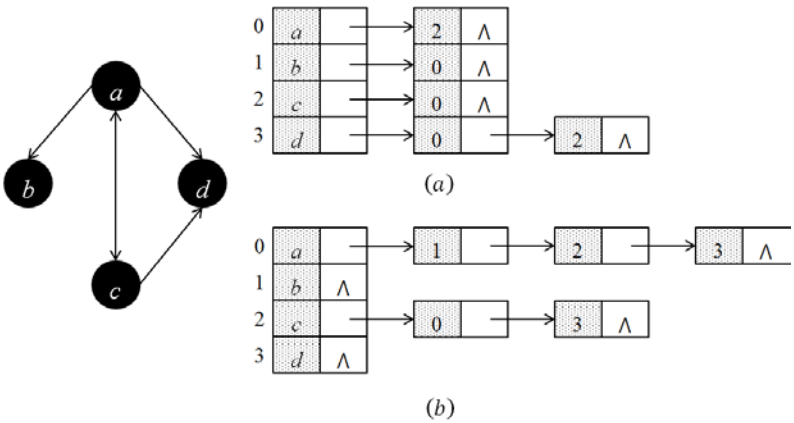


图 8-15　有向图的邻接表

下面给出使用邻接表表示图的存储结构 GraphList 的定义。

```
class EdgeNode;
typedef EdgeNode * EdgeList;

/*边链表*/
class EdgeNode
{
    int vertexNum;                  //边节点在顶点表中的下标
    int weight;                     //权值
    EdgeNode * nextEdgeNode;        //指针字段，指向下一个链节点
};

/*顶点表*/
class Vertex
{
    char vertex;                //顶点信息
    EdgeList edgeList;          //边链表头指针
};

class GraphList
{
    int n;                      //图中顶点个数
    Vertex * vertexes;          //顶点表
};
```

正是因为邻接表对邻接矩阵进行了压缩和优化，忽略了不必要的信息，才能够较大程度地节省存储空间。8.2.3 节将对邻接矩阵和邻接表进行一些比较，从中读者可以更深刻地体会到二者的区别。

8.2.3　两种表示法的比较

邻接矩阵和邻接表是用于表示图这种数据结构的常见存储形式，它们相互关联又存在差异。从空间复杂度角度来分析，很容易推知，一个有n个顶点、e条边的图G的邻接矩阵所需的空间存储代价为$O(n^2)$。如果用邻接表来储存G则存在两种情况。如果G是一个无向图，则该图的邻接表所需的空间代价为$O(n+e)$；如果G是一个有向图，其相

应的空间代价就变为$O(n+2e)$。对于边比较少的稀疏图，即$e \ll n$，则显然使用邻接表来表示图会大大节省空间开销，使得最终的空间代价趋近于$O(n)$。但是如果图中的边较多，特别对于多重图，e接近n^2数量级时，使用邻接表来表示图就不适合了。

对于图有关操作的实现，邻接矩阵和邻接表二者也各有优劣，譬如说要判断图中是否存在一条边(v_i, v_j)，只需要判断其邻接矩阵中第i行第j列的元素是否为 0 即可。而对于邻接表来说，要实现这个功能就不得不遍历顶点v_i的边链表。在这一点上显然邻接矩阵表示法存在优势。而若是想求得图中边的数目，似乎邻接矩阵就落后于邻接表了。对于邻接表来说只需要扫描每个边表就能得出所有的边。但是对于邻接矩阵来说，因为其中的数据量显然多于邻接表，因此要想对矩阵进行一次扫描其时间复杂度是比较高的。

总体上而言，邻接矩阵牺牲的是空间代价，换取的是时间代价，而邻接表恰恰相反。通常在实际应用中也无非是在空间与时间二者之间进行权衡，从而选择合适的结构来解决问题。关于邻接矩阵和邻接表二者之间的一些比较可参照表 8-1。

表 8-1　邻接表和邻接矩阵比较表

	邻接矩阵			邻接表		
		无向图	有向图		无向图	有向图
空间	顶点表节点	n	n	顶点表节点	n	n
	邻接矩阵	$n^2/2$	n^2	边表节点	$2e$	e
	评价	适合稠密图		评价	适合稀疏图	
时间	求顶点的度	扫描矩阵一行 $O(n)$		求顶点的度	扫描某个边表最坏 $O(n)$	
	判断某边存在与否	$O(1)$		判断某边存在与否	扫描某个边表最坏 $O(n)$	
	求边的数目	检测矩阵$O(n^2)$		求边的数目	检测每个边表 $O(n+2e)$	
	建立结构	$O(n^2)$		建立结构	$O(n+e)$	

8.3　图的遍历

从图中某个顶点出发游历图，访遍图中其余顶点，并且使图中的每个顶点仅被访问一次的过程称为图的遍历。当然，遍历图中所有点的前提是该图必须是连通的，本节所讨论的图在默认情况下都认为是连通的。图的遍历算法主要有深度优先和广度优先两种，在不同情况下这两种算法的效率差别很大。但在正式介绍深度优先和广度优先之前，本节会对图中的两个重要问题进行介绍，它们是欧拉路径问题和哈密尔顿路径问题。现实中的很多问题都与这二者有关，或是它们的某种变形。

8.3.1　欧拉路径与欧拉回路

普鲁士的哥尼斯堡镇（现名加里宁格勒）位于普雷格尔河河畔，著名的七桥问题就诞

生在这里。普雷格尔河中有两个小岛，在 18 世纪的时候，河的两岸与小岛之间由 7 座桥连接起来，如图 8-16 所示。

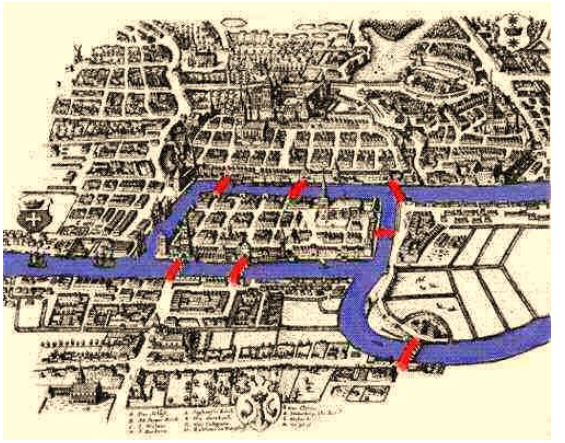


图 8-16　哥尼斯堡七桥

每到傍晚，许多人都来此散步。人们漫步于这 7 座桥之间，久而久之，就形成了这样一个问题：能不能既不重复又不遗漏地一次相继走遍这 7 座桥。这就是闻名遐迩的哥尼斯堡七桥问题。每一个到此游玩或散心的人都想试一试，可是，对于这一看似简单的问题，却没有一个人能符合要求地从 7 座桥上走一遍。

七桥问题也困扰着哥尼斯堡大学的学生们，在屡遭失败之后，他们给瑞士数学家莱昂哈德·欧拉（Leonhard Euler）写了一封信，请他帮助解决这个问题。欧拉看完信后，对这个问题也产生了浓厚的兴趣。他想既然岛和半岛是桥梁的连接地点，两岸陆地也是桥梁的连接地点，那就不妨把这 4 处地方缩小成 4 个点，并且把这 7 座桥表示成 7 条线。这样，七桥问题就抽象地概括成了一道关于图的问题。

遍历图G的每条边一次且仅一次的路径，称为欧拉路径。遍历图G的每条边一次且仅一次的回路，称为欧拉回路。具有欧拉回路的图称为欧拉图。显然若图中存在孤立顶点并不会影响欧拉路径的讨论，因为欧拉路径的要求是遍历图中的每条边而非遍历图中的每个顶点。

欧拉通过分析七桥问题，得到了下面的结论：若无向图中存在欧拉路径，要么只有两个度数为奇数的顶点（也就是仅有起点和终点），要么不存在度数为奇数的顶点。关于这个定理的证明已经超出本书的研究范围，这里不再赘述。

显然七桥问题可以抽象成在图 8-17 中寻找一条欧拉路径的问题。由于七桥问题有 4 个点，且每个点的度都为奇数，所以要找到一条经过 7 座桥，但每座桥只走一次的路线是不可能的。

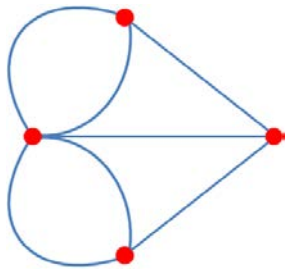


图 8-17　七桥问题

下面我们给出判定一个无向连通图是否为欧拉图的充分必要条件：一个无向连通图是欧拉图，当且仅当该图所有顶点的度数均为偶数。

对于一个有向连通图是否为欧拉图的判定，可以使用下面的定理：一个有向连通图是欧拉图，当且仅当该图的每个顶点的出度等于入度。

读者可以根据上述定理，在图的抽象数据类型中添加欧拉图判定成员方法然后编码实现它，由于这个算法非常简单，本书不再给出源代码。

经典问题求解

文字游戏问题，这也是一道常见的 ACM 竞赛练习题。

最后我们来考虑一个与欧拉路径有关的问题——文字游戏问题。该问题描述如下：一些暗门中隐含着很有趣的文字游戏。一些考古学家要试着破解这些谜题以便将暗门打开。因为除了这个办法之外别无其他选择，所以解开这些谜题对于我们而言是非常重要的。这里每扇门上都有一些具有磁性的盘子。每个盘子上均有一个字母。这些盘子必须按照这样的顺序排列：前一个盘子上单词的最后一个字母是后一个盘子上单词的第 1 个字母。例如，单词 acm 的下一个单词是 motorola。你的任务是编写一个程序去读这些盘子上的文字并判断这些单词是否都能按照以上规则排列，以便把门打开。（题目来源：Central Europe 1999）

输入：该问题的输入由T个测试例子组成。输入文件的第 1 行数据便给出这个数T。每一个测试例子的第 1 行是一个整数N，这个N表示盘子的个数（$1 \leqslant N \leqslant 100000$）。接下来是$N$行数据，每行包含一个单词。每个单词最少由两个，最多由 1000 个小写字母组成。同一个单词可重复出现在列表上。下面给出一段输入示例。

```
3
2
acm
ibm
3
acm
malform
mouse
2
ok
ok
```

输出：程序应能检测盘子是否可以按照这样的顺序排放，即前面盘子上单字的最后一个字母是后面盘子上单字的第 1 个字母。每扇门上的盘子都要用上，并只用一次。如果某单词重复出现n次，那么该重复的单词就需要被使用n次。如果存在这样一个序列，则程序输出信息“ Ordering is possible. ”，否则输出“ The door cannot be opened. ”。

不难看出，该问题可以抽象成在图中寻找一条欧拉路径的问题。利用深度优先对已构建的图进行深度遍历，并判断是否存在欧拉路径即可，下面给出程序实现的源代码。

```
#include <stdio.h>
#include <string.h>
#include "iostream"

using namespace std;

#define N 27

char str[1001];
int v[N];
int is[N][N];

int viii[N];
int dfs(int);
int counter;

int main() {
    int x[N],f[N],j,len,y[N],i,test,flag,lengths,p,q=0;
    int count=0;

    scanf("%d",&test);
    for(i=0;i<test;i++) {
        memset(x,0,sizeof(x));memset(y,0,sizeof(y));
        memset(f,0,sizeof(f));memset(v,0,sizeof(v));
        memset(is,0,sizeof(is));
        for(j=0;j<N;j++)is[j][j]=1;
        scanf("%d",&len);counter=0;
        memset(viii,0,sizeof(viii));
        count=0;

        for(j=0;j<len;j++){
            scanf("%s",str);
            lengths=strlen(str);
            if(str[lengths-1]!=str[0]){
                x[str[0]-'a']++;
                y[str[lengths-1]-'a']++;
                v[str[0]-'a']=v[str[lengths-1]-'a']=1;
                count++;
                is[str[0]-'a'][str[lengths-1]-'a']=1;
                is[str[lengths-1]-'a'][str[0]-'a']=1;
            }
            else f[str[0]-'a']=1;
        }

        for(j=0,p=0,q=0,flag=0;j<N;j++){
            if(x[j]!=y[j]) {
                if(x[j]==y[j]+1) p++;else if(x[j]+1==y[j])q++; else
                    flag++;

            }
```

```
        }

        if(count==0){
            for(j=0;j<N;j++) if(f[j]) count++;
            if(count==1)
                printf("Ordering is possible.\n");
            else
                printf("The door cannot be opened.\n");
            continue;
        }
        for(j=0;j<N;j++) if(v[j]) break;

        dfs(j);
        for(j=0;j<N;j++)
            if(v[j]&&!viii[j]) {j=N+10;break;}
        if(j==N+10) {
                cout<<"The door cannot be opened."<<endl;continue;
        }
        if(!flag&&(p+q==0||p*q==1)) {
            for(j=0;j<N;j++)
                if(f[j]==1&&v[j]!=1) {j=N+10;break;}
            if(j<N+2) printf("Ordering is possible.\n");
            else printf("The door cannot be opened.\n");
        }
        else printf("The door cannot be opened.\n");
    }
    return 0;
}

int dfs(int index) {
    int i=0;
    counter;viii[index]=1;
    for(i=0;i<N;i++)
        if(!viii[i]&&is[i][index]) {
            dfs(i);
        }
    return 0;
}
```

8.3.2 哈密尔顿路径与哈密尔顿回路

小提示

哈密尔顿路径问题还是最早被证明的几个 NP 完全问题之一。对 NP 完全问题感兴趣的读者可以参阅计算理论方面的书籍以了解更多。

1859 年，爱尔兰数学家哈密尔顿（Hamilton）提出下列周游世界的游戏：在正十二面体的 20 个顶点上依次标记伦敦、巴黎、莫斯科等世界著名大城市，正十二面体的棱表示连接这些城市的路线。试问能否在图中做一次旅行，从顶点到顶点，沿着边行走，经过每个城市恰好一次之后再回到出发点。这就是著名的哈密尔顿问题，见图 8-18。

哥尼斯堡七桥问题是在寻找一条遍历图中所有边的简单路径，而哈密尔顿的周游世界问题则是在寻找一条遍历图中所有点的基本路径。在无向图$G = \langle V, E \rangle$中，遍历G中每个顶点一次且仅一次的路径称为哈密尔顿路径，遍历G中每个顶点一次且仅一次的回路称为哈密尔顿回路。具有哈密尔顿回路的图称为哈密尔顿图。哈密尔顿问题是一类问题的总称。

问题提出没过多久，哈密尔顿就收到许多来自世界各地的表明成功周游世界的答案。然而，有没有一个一般的方法来判定一个图是或者不是哈密尔顿图呢？一个半世纪过

去了，这个问题即一个图是否为哈密尔顿图的判定问题至今悬而未决。哈密尔顿问题一直是图论中的世界性难题，到目前为止仍然没有找到一个像欧拉图那样简单的充分必要条件。目前的研究结果仅仅可以分别给出哈密尔顿回路存在的必要条件和充分条件。

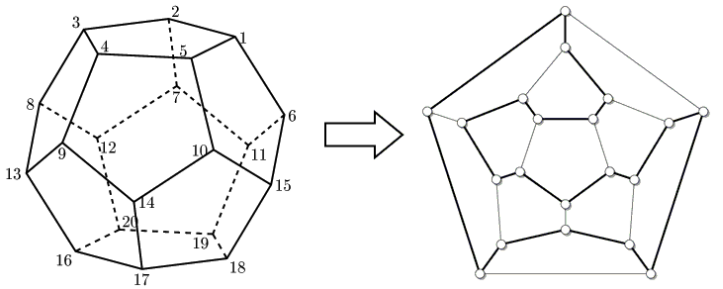


图 8-18　哈密尔顿问题

如果图$G=\langle V,E\rangle$是哈密尔顿图，则它必然具备下述性质。

对V的每个非空真子集S均有$\omega(G-S)\leqslant|S|$，其中$|S|$是S中的顶点数，$\omega(G-S)$表示G删去顶点集S后得到的图的连通分图个数。

上述条件是一个必要条件，但不是一个充分条件。因此，可以运用它来判定某些图不是哈密尔顿图。即哈密尔顿图都满足这样的条件，但符合该条件的图不一定是哈密尔顿图，如图 8-5 所示的彼得森图就不是一个哈密尔顿图，但它却满足上述条件。

下面给出一个判定哈密尔顿回路存在的充分条件。

如果图$G=\langle V,E\rangle$是具有$n\geqslant 3$个顶点的简单无向图，且在图G中每一对顶点的度数和都不小于n，那么G中必然存在一条哈密尔顿回路。

上述条件是一个充分条件，但不是一个必要条件。也就是说满足该条件的图必然是哈密尔顿图，但不满足该条件的图也有可能是哈密尔顿图。

在有向图中也可以定义哈密尔顿有向路径和哈密尔顿有向回路。限于篇幅，本书不做介绍。最后，我们来考虑哈密尔顿回路的一个特例——骑士周游问题。

经典问题求解

骑士周游问题，该问题同时也是使用回溯法进行求解的典型问题。

骑士周游问题（又称骑士漫游或马踏棋盘问题）是算法设计的经典问题。现在已知的关于该问题的最早记录可追溯到公元 9 世纪，印度诗人和文艺理论家楼陀罗托（Rudrata）在其著作《诗庄严论》中对此给出了直观的描述。欧拉是西方最早研究骑士周游问题的数学家之一。但第一个系统化解决骑士周游问题的方法则是由德国学者冯 · 恩斯多夫（H. C. von Warnsdorf）于 1823 年提出的，这个方法现在被称为恩斯多夫规则或恩斯多夫算法。

国际象棋棋盘横向8列，纵向8行，总计64个格子。假设有一个马从棋盘上任意起点出发，要求经过63步之后，不重复地遍历整个棋盘上除初始点以外的每个格子。这个问题用现代语言描述的话，其实就是在一个具体的图中找寻哈密尔顿路径。这里的哈密尔顿路径指的是经过图中每个顶点且只经过一次的一条轨迹。如果该轨迹是一条闭合

路径，即从起点出发不重复地遍历所有点后仍能回到起始点，那么这条路径则称为哈密尔顿回路。图 6-18 显示的是一个马所能跳走的 8 个位置。需要说明的是对于棋盘上任意初始点该问题都有解，且解不唯一。有文献中指出对于一个8×8的棋盘，它拥有的不同巡游路径的个数大约是1.305×10^{35}。

下面给出一些关于骑士周游问题的定理。

定理 1：在8×8棋盘中，令a_{ij}表示第i行第j列的棋格，其中$1 \leqslant i, j \leqslant 8$；$a_{kl}$表示第$k$行第$l$列的棋格，其中其中$1 \leqslant k, l \leqslant 8$。由移动法则可知，从$a_{ij}$到$a_{kl}$是合法的移动，当且仅当$|i-k|=1$，$|j-l|=2$ 或者$|i-k|=2$，$|j-l|=1$。

定理 2：当$|(i+j)-(k+l)|$为偶数时，从a_{ij}到a_{kl}绝不是合法移动；也就是说，只有当$i+j$与$k+l$的奇偶性不同时，a_{ij}到a_{kl}才可能是合法的移动。

考虑将骑士周游问题拓展到$n \times n$的棋盘上，则有以下定理。

定理 3：对于骑士周游问题，当$n \geqslant 5$，且为偶数时，以任意点作为初始点都有解。

骑士周游问题在求解过程中需要用到前面介绍的递归，而且事实上，这还是一个分治与回溯相结合的典型例子。从图 8-19 可知，马每次跳走的选择可能有 8 种，按着其中一种进行尝试，当发现一种情况走不通时再回溯到先前的某一种情况继续尝试其他的选择。如此继续下去即可得到最终的结果。图 8-20 给出了骑士周游问题的两种解的情况，特别其中右图给出的是一个哈密尔顿回路解，即巡游路径的起始点和终结点为同一个棋格。

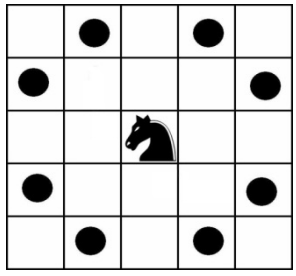


图 8-19　马所能跳走的 8 个位置

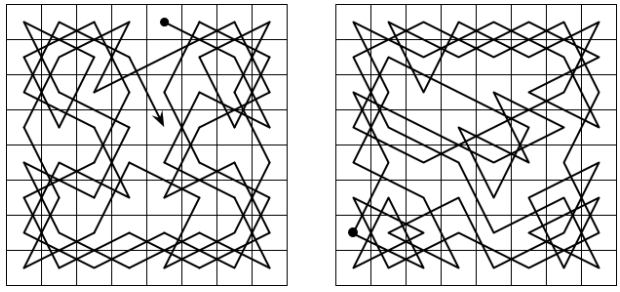


图 8-20　骑士周游问题的两种解

下面考虑编程解决骑士周游问题。前面提到过，欧拉曾经对该问题进行了研究。欧拉

把8×8棋盘上的骑士周游问题转化为在8×8的表格上能否按一定规则排列数字1~64的数学问题。我们的程序也基于这种方法来实现。另外，易知该算法的时间复杂度为$O(8^{N \times N})$，有鉴于如果单纯使用回溯方法而不进行优化，程序效率将是非常低的，于是不妨考虑采用一种启发式的遍历规则：即向前看两步，当每准备跳一步时，设准备跳到(x,y)点，计算(x,y)这一点可能往几个方向跳，将这个数目设为(x,y)点的权值，并将所有可能的(x,y)按权值进行排序，从最小的开始，循环遍历所有可能的(x,y)，回溯求出结果。该算法可以求出所有可能的骑士周游路径，算出一个可行的结果很快。当然，其不足在于时间复杂度本质上并没有改变，所以要求出所有可能结果，仍然很慢。后面的示例程序正是基于这种思想设计完成的。

篇幅所限，本书没有列出执行程序的全部代码，有需要的读者可以从本书的在线支持网站上下载得到完整的源代码。下面给出的是用于执行程序的主函数代码。

```
#include "stdafx.h"
#include "conio.h"
#include <iostream>

using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
    int x,y;
    cout<<"Please input the start point (x,y)."
        <<endl<<"x=";
    cin>>x;
    getchar();
    cout<<"y=";
    cin>>y;
    getchar();
    Board board(x,y);
    board.GetPath();

    return 0;
}
```

编译并运行程序，当输入初始坐标(0,0)时，问题的前两个解如图 8-21 所示。仔细观察这两个解的结果。原来，当经过第63步之后，棋子可以再跳回初始点0的位置，也就是说这两个解不仅仅找到了哈密尔顿通路，而且找到了哈密尔顿回路。这个问题请读者思考，即如何在8×8的棋盘上找到骑士周游问题的哈密尔顿回路解。而且这个结果也是基于前面讨论的启发式方法得到的，观察最初的几个遍历点$0 \to 1 \to 2 \to 3 \to 4 \to 5 \to 6$，刚好是沿着矩形棋盘的两条边走下来的。这是因为2、4、6这些位于边上的位置的试探可能性结果较内部的点少，因此程序会优先遍历它们。

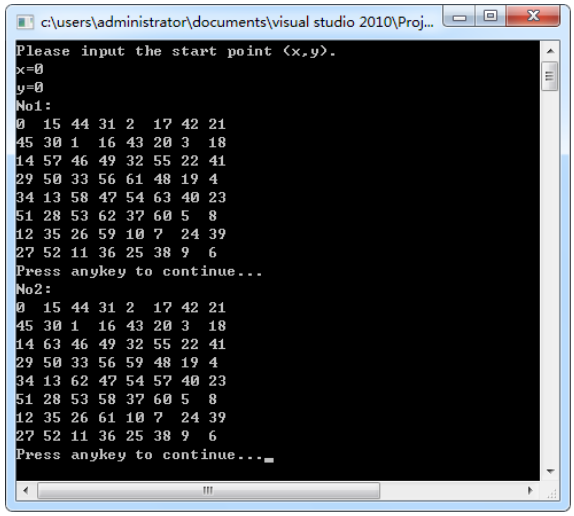


图 8-21　骑士周游程序运行结果

8.3.3　广度优先遍历算法

算法

广度优先遍历算法。

广度优先遍历也称为广度优先搜索，它是一个从起始点出发按照距离递增、逐层访问的方式来遍历图中各点的算法。广度优先搜索可以用来解决可达性问题。换句话说，执行广度优先搜索算法能够回答诸如从起始点出发能否到达目标点这样的问题。这是一个非常重要的应用，因为图中可能包含那些与其他顶点相隔离的顶点，这时就需要对图的连通性进行判定。

广度优先遍历算法以检测从初始点出发的每条边作为开始。当算法检测这些边的时候，新的顶点就会被发现。这些被发现的顶点将被放进一个队列中。然后按顺序检测每个被发现的顶点来决定是否有新的顶点与这些被发现的顶点相关联。在图 8-22 中，广度优先遍历的起始点0被着色显示。

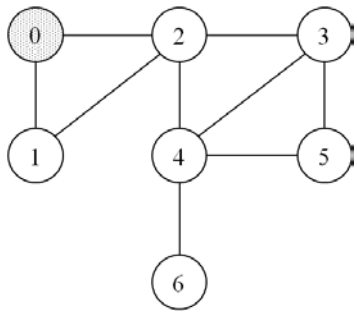


图 8-22　广度优先遍历的起始点

按照上面描述的过程对如图 8-22 所示的图进行遍历。在图 8-23 中，可以看到算法通过检测和顶点0相连的一条边发现了顶点2。这里将顶点2着色，以表示该顶点已经被发现。一旦有一个顶点被发现，就把它放入一个用来存放被发现（但没有被探索的）顶点的队列。所谓被探索，就是指是否检测过该顶点相关联的邻近顶点。然后算法通

过检测剩余的几条边来完成探索起始点的工作，探索顶点0剩余的边使得顶点1被发现。

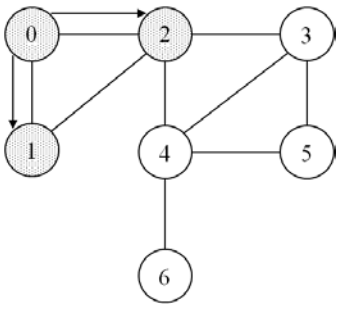


图 8-23　广度优先遍历（1）

一个顶点被完全探索当且仅当它的所有边都被检测过。当一个顶点的探索完成后，算法选择另一个点来进行探索。这个选择不是随意的——被选定的点应当位于被发现但未被探索点队列的首位。在上述例子中就意味着算法接下来需要对2进行探索。如图 8-24 和图 8-25 所示，探索顶点2的过程使得算法发现了顶点4和顶点3。

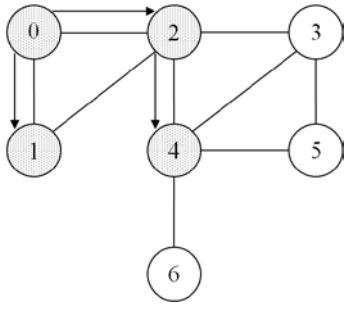


图 8-24　广度优先遍历（2）

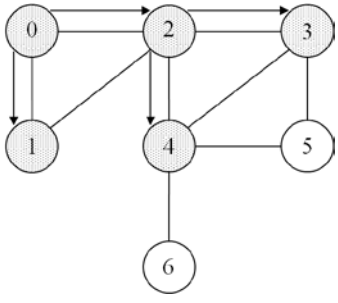


图 8-25　广度优先遍历（3）

在检测了从顶点2出发的所有边之后，算法再考虑顶点1的边。但与顶点1相关的边所含的顶点都已经被发现。所以算法继续进行并考虑与顶点4相连的边。观察可知，与顶点4相连的边有4条。其中两条边所含的顶点已经被发现，剩余的两个未被发现的分别是顶点6和顶点5。然后将它们都标记为已被发现的，并放到队列里。这个过程如图 8-26 和图 8-27 所示。

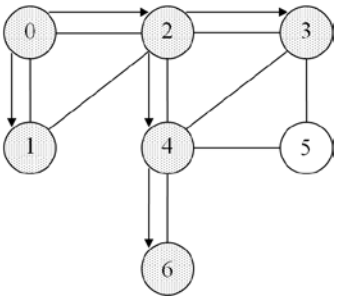


图 8-26　广度优先遍历（4）

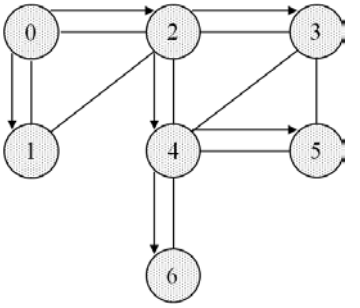


图 8-27　广度优先遍历（5）

至此，算法已经发现了图中所有的顶点。可以看到在图中所有顶点都已被着色。然而算法并不知道这些，因此它会继续检测与顶点3、5和6相连的边。易见，所有这些边所包含的顶点先前都已经被发现了。当所有待探索顶点集为空的时候，算法终止。

广度优先遍历算法的实现代码将在 8.3.4 节中同深度优先遍历算法的代码一并给出。

8.3.4　深度优先遍历算法

算法

深度优先遍历算法。

深度优先遍历又称为深度优先搜索，它是一种非常重要的遍历图的方法。其主要思想是先访问图中某个未被访问过的顶点v_0，并以此为基点选择一个与v_0邻近的但未被访问过的顶点v_1，并访问它。依次深入，顺序访问。若发现某顶点的所有邻近顶点都已经被访问过了，则退回到已访问的顶点中最近的那个，判断它的所有相邻顶点是否都已被访问过，若是则继续回退，否则就访问它的一个未被访问过的相邻顶点v_2，并按同样的方法进行访问。最终访问到图中所有的顶点，实现遍历。

换句话说，深度优先遍历就是探索图中更深的顶点，然后回到接近起始点的点，再继续同样的工作。为了优先探索一个图中更深的部分，深度优先搜索算法必须将被探索过的点保存在一个栈中，而非队列中。这将导致新近发现的顶点，较先前发现的顶点而言，会被优先探索到。在广度优先搜索中，探索顶点的顺序与发现它们的顺序相一致。但是在深度优先搜索中，算法不再这样做，算法会在发现顶点后，马上探索它。

下面举例说明深度优先遍历的工作过程，还是对如图 8-22 所示的图进行操作。如图 8-28 所示，算法以顶点0为出发点进行探索，发现了顶点2。

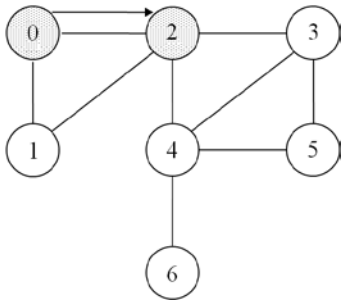


图 8-28　深度优先遍历（1）

算法以顶点2为基点继续开展探索，发现了顶点4，如图 8-29 所示，顶点4被着色。算法继续进行，又发现了顶点6，如图 8-30 所示。

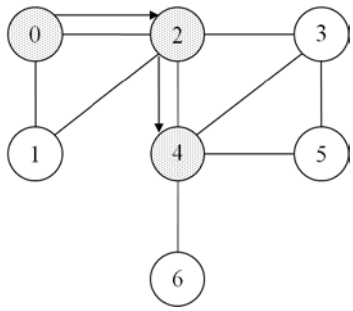


图 8-29　深度优先遍历（2）

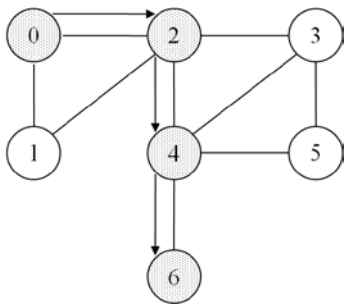


图 8-30　深度优先遍历（3）

当算法发现顶点6的时候，已经不能再向更深处进行探索了，因为所有与顶点6相邻近的顶点都已经被访问过了，因此算法回退到已访问的顶点中最近的那个去检查其是否还有邻近点未被访问过，于是算法发现了顶点5，如图 8-31 所示。

算法以顶点5为基点继续开展探索，发现了顶点3，如图 8-32 所示，顶点3被着色。当算法不能再向更深处进行探索时，算法进行回退，直至退到顶点2，并对顶点2进行探索，于是发现了顶点1。这时算法结束。这个过程如图 8-33 所示。

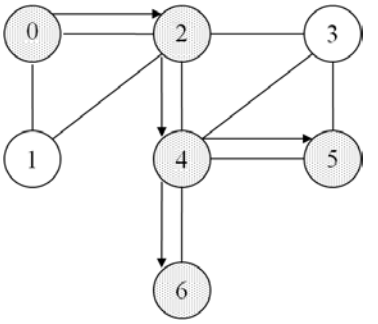


图 8-31　深度优先遍历（4）

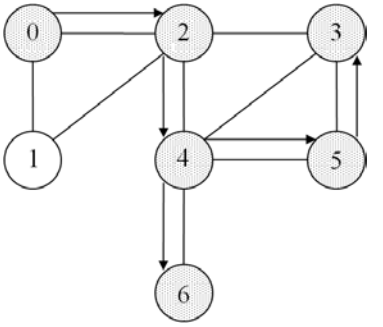


图 8-32　深度优先遍历（5）

小提示

二叉树的遍历算法主要有 3 种，即前序遍历、中序遍历和后序遍历。

树的遍历算法主要有两种，即广度优先遍历和深度优先遍历。

图的遍历算法主要有两种，即广度优先遍历和深度优先遍历。读者应该注意几种遍历算法之间的区别和联系。

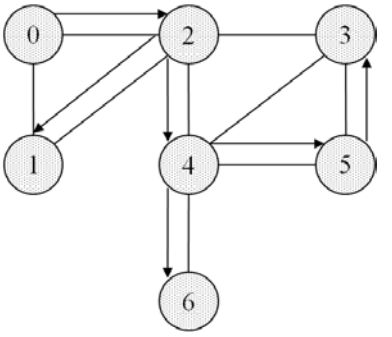


图 8-33　深度优先遍历（6）

根据深度优先搜索的特点，它的实现不同于广度优先，这种不同仅仅体现在所用来存储被探索顶点的容器上。在广度优先搜索中，算法需要按照发现顶点的顺序来探索它们。因此，算法使用队列来维护那些被发现但没有被探索的顶点。深度优先搜索则总是探索那些最近被发现的顶点。为了使这个过程变得简单，算法使用栈而不是堆来存储没有被探索的顶点。

下面给出一个示例程序，它分别使用了深度优先和广度优先两种方法对图进行遍历。

```
#include "stdafx.h"
#include <iostream>
#include <cstdlib>
#include <set>
#include <vector>
#include <list>
```

```
#include <stack>
#include <queue>

using namespace std;

//BFS
void bfs(vector< list<int> >& adj_lists, int start_node) {

    queue<int> not_yet_explored;
    set<int> discovered;

    // 标记起始节点为已被发现，并将其放入队列中开始探索
    not_yet_explored.push(start_node);
    discovered.insert(start_node);

    while (! not_yet_explored.empty()) {

        // 获得一个新节点并依此作为基点进行探索
        int node_to_explore = not_yet_explored.front();
        not_yet_explored.pop();

        // 检测该节点的所有边
        list<int>::iterator edges = adj_lists[node_to_explore].begin();
        for ( ; edges != adj_lists[node_to_explore].end(); edges++) {
            if (discovered.count(*edges) == 0) {
                // 发现新节点将其加入队列
                discovered.insert(*edges);
                not_yet_explored.push(*edges);

                cout <<"Found "<< *edges <<" from "<<node_to_explore<<endl;
            }
        }
    }
}

//DFS
void dfs_helper(vector< list<int> >& adj_lists, set<int>& discovered, int
node) {

    //  检测该节点的所有边
    list<int>::iterator edges = adj_lists[node].begin();
    for ( ; edges != adj_lists[node].end(); edges++) {
        // 检测某条边是否含有未发现的顶点
        if (discovered.count(*edges) == 0) {
            discovered.insert(*edges);
            cout << "Found " << *edges << " from " << node << endl;
            dfs_helper(adj_lists, discovered, *edges);
        }
    }
}

void dfs(vector< list<int> >& adj_lists, int start_node) {

    //标记起始顶点为已被发现
    set<int> discovered;
    discovered.insert(start_node);
    dfs_helper(adj_lists, discovered, start_node);
}
```

```
int _tmain(int argc, _TCHAR* argv[])
{
    //初始化图信息
   vector< list<int> > g(7, list<int>());

   g[0].push_back(2);
   g[0].push_back(1);

   g[1].push_back(0);
   g[1].push_back(2);

   g[2].push_back(4);
   g[2].push_back(3);
   g[2].push_back(0);
   g[2].push_back(1);

   g[3].push_back(2);
   g[3].push_back(4);
   g[3].push_back(5);

   g[4].push_back(6);
   g[4].push_back(5);
   g[4].push_back(3);
   g[4].push_back(2);

   g[5].push_back(4);
   g[5].push_back(3);

   g[6].push_back(4);

   cout << "BFS" <<endl;
   bfs(g, 0);

   cout << endl << "DFS" <<endl;
    dfs(g, 0);

    system("PAUSE");
    return 0;
}
```

读者可以编译并运行上述程序，看看输出结果与本节中所举之例子是否一致。

8.4 最短路径问题

图中从起始点出发到达终止点的路径可能有多条，对于带权图而言，这些路径的长度可能不同，其中路径长度最短的那条路径称为两个顶点间的最短路径。最短路径的求解是一个具有现实意义的课题，例如火车线路的选择问题等都需要应用最短路径方面的结论和方法来进行求解。

8.4.1 固定起点最短路径问题

最短路径问题可分为固定起点最短路径问题和所有顶点间最短路径问题，其中固定起点最短路径更具有普遍意义。固定起点最短路径问题的求解算法有很多种，针对不同的网络特征、应用需求及具体的软硬件环境，各种算法在空间复杂度、时间复杂度以

及易实现性等方面各具特色。其中，采用贪心式启发策略设计的狄克斯特拉算法是目前已知的理论上最完善的算法，该算法是由荷兰计算机科学家艾兹格·狄克斯特拉（Edsger Wybe Dijkstra）于 1959 年提出的。

狄克斯特拉算法是基于这样一个事实设计的，即“最短路径的任一段子路径都是最短路径。”从这个原理出发，算法在实现上总是对图中的每个顶点存有两个标记，算法的每一步都是在对这些标记进行修改，从而完成最短路径的选择任务。

算法

最短路径的狄克斯特拉算法，因由大名鼎鼎的狄克斯特拉提出而得名。狄克斯特拉还提出了信号量与 PV 原语，解决了著名的哲学家聚餐问题，他同时也是“goto”有害论的提出者。狄克斯特拉曾于 1972 年获得计算机领域的最高奖——图灵奖，他与克努特并称为我们这个时代最伟大的两位计算机科学家。

下面给出利用狄克斯特拉算法求解带权图$G(V,E)$中从顶点u_0到其余各顶点最短路径的操作步骤。首先来定义几个变量。

集合U是标号顶点集合，当从初始顶点到某点的最短路径被求得后，就将这个顶点放入该集合内。算法开始时对U进行初始化，令$U=\{u_1\}$。

标记$l(v)$用于记录起始顶点u_1到另一顶点v的路径的长度，并规定当$v=u_1$时，有$l(v)=0$，这时u_1到其他各顶点的长度均为无穷远。

标记$z(v)$记录的是获得从起始顶点u_1到另一顶点v之间最短路径中v的前一个顶点，它可以用来确定最短路径的路线。算法开始时，$z(v)=u_1$。

初始化完成后，算法检测从U中任意点到$V-U$中任意点存在的路径，如果尚有顶点不可达，则规定其路径长度为无穷远。如果存在任意$v\in V-U$，有$l(v)>l(u)+W(u,v)$，则令$l(v)=l(u)+W(u,v)$。然后检测所有$l(v)$中最小的，对于使$l(v)$的值最小的顶点v，将其标记后并入集合U。最后更新$z(v)=u$。前面提过，算法的核心就是不断更新$l(v)$和$z(v)$，因此这一步十分关键。更新工作完成后，算法会进入下一步迭代，直到$V-U$为空时，最短路径寻找完毕，算法终止。

狄克斯特拉算法对于初学者而言是一个难点。下面举例说明该算法是如何工作的，这里以图 8-2 为例，求从顶点1到其他各顶点的最短路径。首先写出该图的邻接矩阵如下：

$$\boldsymbol{W}=\begin{bmatrix}0 & 10 & 21 & \infty & 8 & \infty\\ 10 & 0 & 18 & 5 & \infty & 6\\ 21 & 18 & 0 & \infty & 25 & 19\\ \infty & 5 & \infty & 0 & \infty & 7\\ 8 & \infty & 25 & \infty & 0 & 33\\ \infty & 6 & 19 & 7 & 33 & 0\end{bmatrix}$$

算法初始化$U=\{u_1\}$，此时u_1到其他各顶点的长度均为无穷远，则标记$l(v=u_1)=0$以及$z(v=u_1)=u_1$。

算法继续进行，此时发现从U中任意点到$V-U$中任意点存在的路径有：(u_1,u_2)、(u_1,u_3)和(u_1,u_5)，除了u_2、u_3和u_5的$V-U$中其他顶点均不可达，因此路径长度为无穷远。注意到$l(u_2)$的值在上一轮中等于无穷大，所以有$l(u_2)=\infty>l(u_1)+W(u_1,u_2)=10$，于是更新$l(u_2)=10$，同理更新$l(u_3)=21$，$l(u_5)=8$。可见其中路径长度最小的是$l(u_5)=8$，所以标记顶点$u_5$，将其加入$U$中，$z(v=u_5)=u_1$。

迭代到第 3 轮时，$U=\{u_1,u_5\}$，$V-U=\{u_2,u_3,u_4,u_6\}$，从U中任意点到$V-U$中任意

点存在的路径有：(u_1,u_2)、(u_1,u_3)和(u_5,u_6)，其他顶点均不可达，路径长度为无穷远。其中，$l(u_6)=\infty>l(u_5)+W(u_5,u_2)=41$，所以更新$l(u_6)=41$。同时有$l(u_2)=10$以及$l(u_3)=21$, 可见其中路径长度最小的是$l(u_2)=10$, 所以标记顶点$u_2$，将其加入$U$中，即$z(v=u_2)=u_1$。

迭代到第 4 轮，$U=\{u_1,u_2,u_5\}$，$V-U=\{u_3,u_4,u_6\}$，从U中任意点到$V-U$中任意点存在的路径有：(u_1,u_3)、(u_2,u_4)、(u_2,u_6)和(u_5,u_6)。其中，$l(u_4)=\infty>l(u_2)+W(u_2,u_4)=15$，所以更新$l(u_4)=15$。此外，$l(u_6)=41>l(u_2)+W(u_2,u_6)=16$，于是也更新它。目前，$l(u_4)=15$，$l(u_3)=21$，$l(u_6)=16$，可见其中路径长度最小的是$l(u_4)=15$，所以标记顶点$u_4$，将其加入$U$中，$z(v=u_4)=u_2$。

如此继续迭代下去直到$V-U$为空，最短路径寻找完毕，算法终止。整个过程如表 8-2 所示。

表 8-2　狄克斯特拉算法步骤

	u_1	u_2	u_3	u_4	u_5	u_6
1	0	∞	∞	∞	∞	∞
2		10	21	∞	8	∞
3		10	21	∞		41
4			21	15		16
5			21			16
6			21			
$l(v)$	0	10	21	15	8	16
$z(v)$	u_1	u_1	u_1	u_2	u_1	u_2

经典问题求解

旅游交通路线问题，背包客必备！

下面利用狄克斯特拉算法来解决一个实际问题——旅游交通路线问题。塔斯马尼亚岛位于澳洲大陆东南海面上，西临印度洋，东濒太平洋的塔斯曼海，北隔深不及 60m 的巴斯海峡与澳大利亚大陆相望，距离墨尔本南部约 240km。塔斯马尼亚是澳大利亚最小的州，是唯一一个人们可以用几天就能转一圈的州。岛上风光独特，连绵的丘陵、山谷、高原、火山和陡峭的海岸，令人神往。塔斯马尼亚岛是著名的观光旅游胜地，素有旅游之州的美誉。

图 8-34 为一幅塔斯马尼亚岛的地图，如图所示，岛上有 8 个地点，游客可以驱车沿公路游玩该岛。但需注意的问题是从某点出发不一定可以到达图中的所有地点，某些时候，公路可能因为雪、泥石流、交通事故等原因而关闭。此外，一方面游客需要考虑选择一条距离尽可能短的路线来到达目的地；另一方面，由于沿途路况和其他因素的影响，每条旅行路线所需的旅行费用也不同。希望读者能够编写一个程序，该程序接收用户输入的起始城镇名称和终点城镇名称，然后经计算后给出一条达到目标城市的最便宜的线路，并给出途经的每一个城镇和总共所需行进的路程。即该程序能为用户选择一条通向目的地的最廉价的路线，同时提供该路线途径的地点和总的距离。

这里首先提供一个名为“ MapInformation.txt ”的文本文件，该文件中的信息显示了当前公路的状态。这个文件由若干行组成，每行代表带权图中的一条边。每行由 4 部分

组成，它们分别是c_0、c_1、fee以及distance，中间用空格隔开。其中c_0和c_1是一条公路连接的两个城镇的名称，fee代表该路线的旅行花费，distance代表该条线路的长度。文件的具体内容请读者到本书的在线支持网站上下载，这里不详细列出。

图 8-34　塔斯马尼亚岛地图

该程序将利用 3 个类来解决问题，它们是 City 类、Road 类和 RoadSystem 类。City 类包含所有城市的信息。Road 类模拟公路交通系统中的一条路段。这个类包括的共有成员变量用以表示目的城镇、路费和距离。以上两个类都会被 RoadSystem 类使用。下面给出 City.h 文件的代码清单，其中包含了 City 类的定义和实现。

```
#ifndef _CITY_H_
#define _CITY_H_

using namespace std;

class City {

public:

    string name;       // 城镇的名称
    bool  visited;
    int total_fee;
    int total_distance;
    string from_city;

     //默认构造函数
    City() : name(""), visited(false), total_fee(0),
                 total_distance(0), from_city("") {}

    City(string const &s): name(s), visited(false),
    total_fee(0), total_distance(0), from_city("") {}
};

#endif
```

下面给出 Road.h 文件的代码清单，其中包含了 Road 类的定义和实现。

```
#ifndef _ROAD_H_
#define _ROAD_H_

#include <string>

using namespace std;

class Road {

public:

   int fee;
   int distance;
   string destination;

   Road(string city, int f, int d) : fee(f),
   distance(d), destination(city) {}
};

#endif
```

该问题是一个典型的最短路径问题，可以使用狄克斯特拉算法进行求解。为了组织算法所需的数据结构，程序必须创建一个带权图来存储 MapInformation.txt 所提供的信息。其中每个城市都是图中的一个节点，每条路都是图中的一条边。与边相关的权值有两个，分别是路费和路程。程序将求得一条从初始城市到达目的城市的最便宜的路径。RoadSystem 类模拟了该公路交通系统，它使用邻接表表示法存储图信息。

这些邻接表使用 STL 库中的 map 类来辅助实现，结构如图 8-35 所示。该邻接表通过一个字符串类型到一个 list<Road*>类型的映射来表示公路系统。图 8-35 显示了被表示的公路系统的一部分。注意为了简化图形，路费和路程信息被省略了。

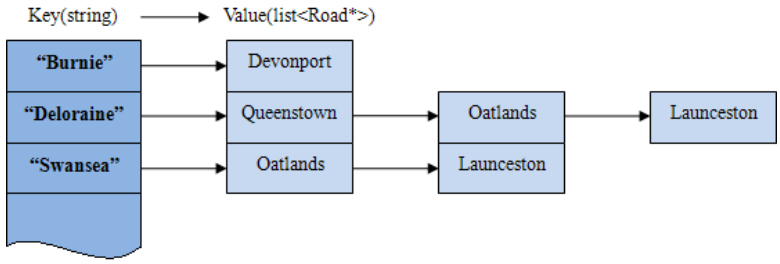


图 8-35　邻接表

首先对 RoadSystem 类中的几个主要的成员函数进行说明。

- load_roads()

它通过读取 MapInformation.txt 文件所提供的信息来建立存储图信息的邻接表。

- reset()

这个函数遍历图中的所有城市然后重置 City 对象的数据成员。

- ~RoadSystem()

该析构函数确保所有城市和道路的信息被有效释放。

- calc_route()

该函数接收起始城市名称和终止城市名称作为参数，使用狄克斯特拉算法进行处理，并最终返回一个 pair<int, int> 类型的对象，其中包含了从起始城市到达目的城市的总费用和距离。若从起始城市到目的城市之间没有一条有效路径就返回 pair(INT_MAX, INT_MAX)。

- recover_route()

该函数接收一个目的城市的名称作为参数，并返回一个字符串类型的变量，其中的内容是从起始城市到目的城市的一条费用最低的路径，表现形式是将途径的城市名称用一个 4 字符的字符串“ -> ”连接起来。

在狄克斯特拉算法实现中，为了完成一个新的搜索，一个用于表示起始城市的指向 City 对象的指针应当被加入到一个初始为空的候选者队列中。算法继续探索由每个候选城市出发的所有公路，并有可能向候选队列中加入新的城市。用一个数字标记了每个 City 对象，该数字表示到目前为止，从初始城市到达该城市的一条最便宜的路径所需的花费。它被存在 City 类的 total_fee 成员变量中。这个值会随着新的更便宜的路径的发现而变小。

为了准确地记录搜索工作求得的路径，每一个 City 对象都包含一个字符串类型的成员变量 from_city。该字符串将由搜索算法负责更新，它总是包含着一个城市的名称，经由该城市所得到的路径就是费用最低的路径。当搜索工作完成，费用最低的路径也随之获得。

限于篇幅，这里将不罗列全部的代码文件，读者可以从本书的在线支持网站上下载得到 RoadSystem.h 文件和下面给出的 RoadSystem.cpp 文件的代码清单，它们实现了 RoadSystem 类的所有成员函数。最后，我们给出 Main.cpp 文件的代码清单，它包含了驱动整个程序运行的主函数。

```
#pragma warning (disable:4786)
#pragma warning (disable:4503)

#include <iostream>
#include <fstream>
#include <string>
#include <list>
#include <map>
#include <queue>

#include "City.h"
#include "Road.h"
#include "RoadSystem.h"

using namespace std;
```

```
int main() {

   try {
      RoadSystem rs("MapInformation.txt");
      while (true) {
         cerr << endl << endl
               <<"请输入起点城市和终点城市: (退出请输入'quit')"<<endl;
         string from, to;
         cin >> from;
         if (from == "quit") break;
         cin >> to;

         if (rs.is_valid_city(from) && rs.is_valid_city(to)) {
            rs.output_cheapest_route (from, to, cout);
         }
         else {
            cout << "无此城市, 请确认后重试!"<<endl<<endl;
         }
      }
      return EXIT_SUCCESS;
   }
   catch (exception& e) {
      cerr << e.what() << endl;
   }
   catch (...) {
      cerr << "程序出现异常, 请退出程序后重试。"<<endl;
   }

   return EXIT_FAILURE;
}
```

请读者完成编码后，编译并运行程序。

8.4.2 非固定起点最短路径问题

算法

弗洛伊德算法，因由计算机科学家弗洛伊德提出而得名。弗洛伊德还是断言法和堆排序算法的创始人。因其在设计高效可靠软件之方法学方面的杰出贡献，弗洛伊德于1978年荣获图灵奖。

狄克斯特拉算法用于求解固定起点最短路径问题，也称单源最短路径问题，也就是说每使用一次该算法仅能求得图中固定的一点到达其他各点的最短路径。对于有n个顶点的带权图而言，如果希望求得图中任意两点之间的最短路径，可以尝试调用n次狄克斯特拉算法。由于狄克斯特拉算法的时间复杂度是$O(n^2)$，所以调用n次的时间复杂度就是$O(n^3)$。除了这种方法是否还存在其他求解图中任意顶点间最短路径的算法呢？答案是肯定的。本节就介绍其中一种利用动态规划思想设计的（也是最常被用到的）方法——弗洛伊德算法，该算法由美国计算机科学家罗伯特·弗洛伊德（Robert Floyd）于1962年提出，它的时间复杂度为$O(n^3)$，但执行过程较调用n次狄克斯特拉算法的方案来说就简单许多了。

弗洛伊德算法的核心思想是以图的邻接矩阵为基础，通过不断在图中插入新顶点的方式来更新邻接矩阵中任意两点间的最短路径，从而求得图中任意顶点间的最短路径。除了用来计算最短路径的距离邻接矩阵之外，算法还需要用一个同等大小的辅助矩阵来记录最短路径的路由线路，该矩阵称为路由矩阵。

这里首先介绍距离矩阵的生成方法。对于具有n个顶点的带权图$G(V,E)$，首先把该带

权图的邻接矩阵作为距离矩阵的初始状态。即：

$$\boldsymbol{D}^{(0)} = [d_{ij}^{(0)}]_{n\times n} = \boldsymbol{W}$$

然后将顶点v_1插入并更新距离矩阵，即在考虑仅有顶点v_1存在的情况下，将顶点v_1作为中间点，用从顶点v_i到顶点v_j的最短距离更新原距离矩阵，即

$$\boldsymbol{D}^{(1)} = \left[d_{ij}^{(1)} = \min\{d_{ij}^{(0)}, d_{i1}^{(0)} + d_{1j}^{(0)}\}\right]_{n\times n}$$

算法继续进行，将顶点v_2插入并更新距离矩阵，在考虑仅有顶点v_1、v_2存在的情况下，将顶点v_1和v_2作为中间点，用从顶点v_i到顶点v_j的最短距离更新原距离矩阵，即

$$\boldsymbol{D}^{(2)} = \left[d_{ij}^{(2)} = \min\{d_{ij}^{(1)}, d_{i2}^{(1)} + d_{2j}^{(1)}\}\right]_{n\times n}$$

如此继续下去，不断重复插入顶点、更新矩阵的步骤，直到图中的所有顶点都已插入完毕为止。

下面介绍路由矩阵的生成方法。路由矩阵$\boldsymbol{R}$是一个与距离矩阵$\boldsymbol{D}$同等大小的矩阵，可以用$\boldsymbol{R} = (r_{ij})_{n\times n}$来表示。$r_{ij}$表示从顶点$v_i$到顶点$v_j$的最短路径需要经过点号为$r_{ij}$的顶点。算法开始时初始化。

$$\boldsymbol{R}^{(0)} = [r_{ij}^{(0)}]_{n\times n}, \qquad r_{ij}^{(0)} = j$$

当某顶点v_k被插入时，距离矩阵中的某个元素d_{ij}被更新。那么相应地，在路由矩阵中，r_{ij}也被更新为k。更新规则可以用如下公式表示。

$$r_{ij}^{(k)} = \begin{cases} k, & d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \\ r_{ij}^{(k-1)}, & d_{ij}^{(k-1)} \leqslant d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \end{cases}$$

为了帮助读者更好地理解弗洛伊德算法的具体操作步骤，下面举例说明该算法的工作过程。这里以图 8-2 为例，求其任意顶点之间的最短路径，首先初始化图的邻接矩阵和路由矩阵如下。

$$\boldsymbol{D}^{(0)} = \begin{bmatrix} 0 & 10 & 21 & \infty & 8 & \infty \\ 10 & 0 & 18 & 5 & \infty & 6 \\ 21 & 18 & 0 & \infty & 25 & 19 \\ \infty & 5 & \infty & 0 & \infty & 7 \\ 8 & \infty & 25 & \infty & 0 & 33 \\ \infty & 6 & 19 & 7 & 33 & 0 \end{bmatrix} \quad \boldsymbol{R}^{(0)} = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 1 & 2 & 3 & 4 & 5 & 6 \\ 1 & 2 & 3 & 4 & 5 & 6 \\ 1 & 2 & 3 & 4 & 5 & 6 \\ 1 & 2 & 3 & 4 & 5 & 6 \\ 1 & 2 & 3 & 4 & 5 & 6 \end{bmatrix}$$

当插入顶点v_1时，对矩阵$\boldsymbol{D}^{(0)}$进行更新，观察可知顶点2和顶点5之间的距离由原来的无穷远缩小到18，所以将d_{25}和d_{52}更新为18。同时，更新路由矩阵$\boldsymbol{R}^{(0)}$。将其中的r_{25}和r_{52}更新为1。更新结果如下。

$$\boldsymbol{D}^{(1)} = \begin{bmatrix} 0 & 10 & 21 & \infty & 8 & \infty \\ 10 & 0 & 18 & 5 & \boxed{18} & 6 \\ 21 & 18 & 0 & \infty & 25 & 19 \\ \infty & 5 & \infty & 0 & \infty & 7 \\ 8 & \boxed{18} & 25 & \infty & 0 & 33 \\ \infty & 6 & 19 & 7 & 33 & 0 \end{bmatrix} \quad \boldsymbol{R}^{(1)} = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 1 & 2 & 3 & 4 & \boxed{1} & 6 \\ 1 & 2 & 3 & 4 & 5 & 6 \\ 1 & 2 & 3 & 4 & 5 & 6 \\ 1 & \boxed{1} & 3 & 4 & 5 & 6 \\ 1 & 2 & 3 & 4 & 5 & 6 \end{bmatrix}$$

当插入顶点v_2时，对矩阵$\boldsymbol{D}^{(1)}$进行更新，观察可知顶点1和顶点4之间的距离由原来的无穷远缩小到15，顶点1和顶点6之间的距离由原来的无穷远缩小到16，顶点3和顶点4之间的距离由原来的无穷远缩小到23。所以由此分别更新距离矩阵中的各项。此外，顶点4和顶点5之间的距离，经由路径$5 \to 1 \to 2 \to 4$缩小到23；顶点5和顶点6之间的距离，经由路径$5 \to 1 \to 2 \to 6$由原来的33缩小到24，因此也对它们进行更新。路由矩阵$\boldsymbol{R}^{(1)}$更新至$\boldsymbol{R}^{(2)}$。

$$\boldsymbol{D}^{(2)}=\begin{bmatrix} 0 & 10 & 21 & \boxed{15} & 8 & \boxed{16} \\ 10 & 0 & 18 & 5 & 18 & 6 \\ 21 & 18 & 0 & \boxed{23} & 25 & 19 \\ \boxed{15} & 5 & \boxed{23} & 0 & \boxed{23} & 7 \\ 8 & 18 & 25 & \boxed{23} & 0 & \boxed{24} \\ \boxed{16} & 6 & 19 & 7 & \boxed{24} & 0 \end{bmatrix} \quad \boldsymbol{R}^{(2)}=\begin{bmatrix} 1 & 2 & 3 & \boxed{2} & 5 & \boxed{2} \\ 1 & 2 & 3 & 4 & 1 & 6 \\ 1 & 2 & 3 & \boxed{2} & 5 & 6 \\ \boxed{2} & 2 & \boxed{2} & 4 & \boxed{2} & 6 \\ 1 & 1 & 3 & \boxed{2} & 5 & \boxed{2} \\ \boxed{2} & 2 & 3 & 4 & \boxed{2} & 6 \end{bmatrix}$$

在本例中，当插入顶点v_3、v_4、v_5、v_6时，距离矩阵不再发生更新，最终距离矩阵和路由矩阵如下。

$$\boldsymbol{D}^{(3)}=\boldsymbol{D}^{(4)}=\boldsymbol{D}^{(5)}=\boldsymbol{D}^{(6)}=\begin{bmatrix} 0 & 10 & 21 & 15 & 8 & 16 \\ 10 & 0 & 18 & 5 & 18 & 6 \\ 21 & 18 & 0 & 23 & 25 & 19 \\ 15 & 5 & 23 & 0 & 23 & 7 \\ 8 & 18 & 25 & 23 & 0 & 24 \\ 16 & 6 & 19 & 7 & 24 & 0 \end{bmatrix}$$

$$\boldsymbol{R}^{(3)}=\boldsymbol{R}^{(4)}=\boldsymbol{R}^{(5)}=\boldsymbol{R}^{(6)}=\begin{bmatrix} 1 & 2 & 3 & 2 & 5 & 2 \\ 1 & 2 & 3 & 4 & 1 & 6 \\ 1 & 2 & 3 & 2 & 5 & 6 \\ 2 & 2 & 2 & 4 & 2 & 6 \\ 1 & 1 & 3 & 2 & 5 & 2 \\ 2 & 2 & 3 & 4 & 2 & 6 \end{bmatrix}$$

最后可以从距离矩阵$\boldsymbol{D}^{(5)}$中得到图中各顶点间的最短路径长度。例如，由于$d_{56}=24$，所以顶点5和顶点6之间的最短路径长度为24。

此外，还可以从路由矩阵$\boldsymbol{R}^{(5)}$中得到该路径的路由线路。仍以上面的问题为例，若希望求得顶点6到顶点5的最短路径的路由线路，观察矩阵可知$r_{65}=2$，这表示路由线路中顶点6的下一个顶点是2，因此由$r_{25}=1$，可知顶点2的下一个顶点是1，再由$r_{15}=5$，可知1的下一个顶点是5。所以从顶点6到顶点5的最短路径为：$6 \to 2 \to 1 \to 5$。

算法

最短路径的动态规划算法。

8.4.3 最短路径的动态规划解法

动态规划是一种非常复杂的算法设计思想，它是现实生活和生产中用于解决实际问题的最常用方案之一。作为运筹学的一个分支，动态规划广泛应用于工程技术、经济管理、生产调度和控制工程中，为解决某些实际问题而发挥着巨大的作用。由于动态规划的内容较多且相对复杂，因此本书无法对其进行一个非常具体而全面的介绍，本节将在其相关概念、理论和方法的简要介绍之后给出使用动态规划求解最短路径的具体方法和步骤。

现实中很多问题的完成过程常包含许多不同的阶段，这种问题具有多阶段性，我们称其为阶段决策过程。例如，多段图中的最短路径问题：在图 8-36 中找出从A_1到D_1的最短路径。

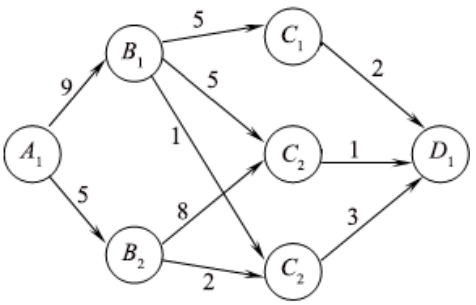


图 8-36　多段图

观察图 8-36，易见图中的点被分为 4 类（A、B、C、D），图中所有的边都处于相邻的两类点之间，并且都从前一类点指向后一类点。这样一来，图中的边就被分成了 3 类（$A \to B$、$B \to C$、$C \to D$）。该问题需要从每一类中选出一条边来，组成从A_1到D_1的一条路径，并且这条路径是所有这样的路径中最短的。这就是一个典型的多阶段决策过程。准确地说，多阶段决策过程就是指一类特殊的活动过程，其过程本身可以按某种顺序分解成若干相互联系的阶段，在每一个阶段都要做出决策，全部过程的决策是一个决策序列。要使整个活动的总体效果达到最优的问题，称为多阶段决策问题。

多阶段决策过程的互不相同且又彼此联系的各阶段中，原问题往往会被发散成多个子问题，子问题的数目可能无法提前获得，子问题之间又可能相互交织，因此在每个阶段都需要做出相应的决策，也就是舍弃某些不大可能的路径而专注于最优解。各个阶段的决策依赖于当前状态又对后续状态产生影响，因此各个阶段的决策都应遵循一定的原则和规律。由于多阶段性和子问题众多，常规问题求解方法的运算规模可能是非常庞大的，这时就需要用到动态规划的方法来解决问题。

动态规划是解决多阶段决策过程最优化问题的一种数学方法，它采用最优化原理来建立用于计算最优解的递归表达式。最优化原理是动态规划的理论基础，它是由美国数学家理查德·贝尔曼（Richard Ernest Bellman）等人于 20 世纪 50 年代提出的。1951 年，贝尔曼最先提出了解决多阶段决策过程最优化问题的“最优化原理”，6 年后，他发表了名著《动态规划》，该书是动态规划领域的第一本著作。此后，动态规划方法在许多方面都得到了广泛的应用，取得了显著的效果。

最优化原理的基本内容是，“作为整个过程的最优策略具有这样的性质：即无论过去的状态和决策如何，对前面的决策形成的状态而言，余下的诸决策必须构成最优策略。”尽管最优化原理是动态规划的理论基础，但是随着研究的深入，人们发现它仅仅是策略最优性的一个必要条件，托马斯·莫棱（Thomas Morin）在 1982 年证明了这一点。后来，学者们提出了条件更强的最优性定理，最优性定理是动态规划方程的直接反映，它是策略最优性的充分必要条件。该定理描述如下：设阶段数为n的多阶段决策过程，

其阶段编号依次为$k=0,1,2,\cdots,n-1$。允许策略$P_{0,n-1}^{*}=(x_0^{*},x_1^{*},\cdots,x_{n-1}^{*})$是最优策略的充要条件是对任意一个$k$和初始状态$s_0$（其中$0<k<n-1$，$s_0\in S_0$，）有下式成立。

$$V_{0,n-1}(s_0,P_{0,n-1}^{*})=\underset{P_{0,k-1}\in P_{0,k-1}(s_0)}{\text{opt}}\left\{V_{0,k-1}(s_0,P_{0,k-1})+\underset{P_{0,k-1}\in P_{0,k-1}(s_0)}{\text{opt}}\left[V_{k,n}(\widetilde{s_k},P_{k,n-1})\right]\right\}$$

其中，$P_{0,k-1}=(P_{0,k-1},P_{k,n-1})$，$\widetilde{s_k}=T_{k-1}(s_{k-1},x_{k-1})$，它是由初始状态$s_0$和子策略$P_{0,k-1}$所确定的$k$阶段状态，$T_{k-1}(s_{k-1},x_{k-1})$是状态转移方程。

不难看出，最优化原理是最优性定理的一个推论，因此最优性定理的实用性更广，更能够揭示动态规划方法的本质。

动态规划是一种方法，是考虑问题的一种途径，动态规划并不是一种算法。正如前面所讲过的回溯法、贪婪算法和分治法一样，它无法提供一套固定的求解模式或步骤，它是针对具体问题构建具体算法的一种思想。有效地应用动态规划方法通常需要很强的逻辑思维和建模能力，也正是因为这样，动态规划一直都是国内外信息学竞赛中最热的考点。

动态规划同前面曾经讲述过的许多算法设计思想之间存在交叉和联系，下面我们就从联系与区别这个角度来讨论一下动态规划的本质。

首先，动态规划和贪婪算法有一定的相似性，它们都可将一个问题的解决方案视为一系列决策的结果。但不同的是贪婪算法的当前选择可能要依赖已经做出的所有选择，但不依赖于有待于做出的选择和子问题。因此贪婪算法呈现出一种自顶向下的求解过程，且每次采用贪婪准则所做出的决策都具有不可撤销性。因此，贪婪算法相对于动态规划的不足就体现在难以通过局部的贪婪策略达到全局最优解。在动态规划中，算法会考察每个最优决策序列中是否包含一个最优子序列。也就是说，动态规划把某个问题分成若干个阶段后，得到最优解的过程都包含若干个从以前的阶段通过最优策略进行转移的过程。动态规划从这一点上来保证结果的最优性。

其次，由于动态规划是一种将问题实例分解为更小的、相似的子问题，并存储子问题的解而避免计算重复的方法，因此动态规划本质上体现了分治法的思想。但不同的是分治法要求各个子问题都是独立的，一旦子问题是相互关联并交叉的，那么分治策略就势必要做许多不必要的工作。这种重复求解公共的子问题的过程由于缺乏有效的优化而最终导致求解效率十分低下。动态规划在对付重复出现的子问题时，只在第一次遇到时加以求解，然后把答案保存起来，以便再次遇到时使用，从而避免了重复无用的工作。

在向读者介绍应用动态规划进行问题求解的基本步骤之前，这里先定义一些常用的术语。

（1）阶段

将所给的问题恰当地分成若干个相互联系、互不交叉的阶段，以便求解，一般用k表示阶段变量。

（2）状态及状态变量

状态是某阶段的初始位置，同时也是前一阶段的终点，通常一个阶段会包含有若干个状态，用于描述状态的变量称为状态变量。状态变量可以是一个变量，也可以是一组变量的集合。状态变量的个数也反映出状态的维数情况。常用s_k表示第k阶段的状态变量，状态变量s_k的取值集合称为状态集合，用S_k表示。

（3）决策及决策变量

决策是在某阶段状态给定后，从该状态演变到下一个阶段某一状态的选择。动态规划过程通过决策来确定下一个阶段的状态。用于描述决策的变量称为决策变量。通常用$x_k(s_k)$来表示第k阶段当状态为s_k时的决策变量，可见决策变量是状态的函数。此外，可以用$D_k(s_k)$来表示第k阶段状态为s_k时的决策集合，并有$x_k(s_k) \in D_k(s_k)$。

（4）状态转移方程

状态转移方程描述由一个状态到另一个状态的演变过程。即在多阶段决策过程中，当第k阶段的决策变量已经给定，由第k阶段到第$k+1$阶段的状态转移规律就是所谓的状态转移方程。通常用$s_{k+1} = T(s_k, x_k)$表示由第k阶段到第$k+1$阶段的状态转移方程。

（5）策略与子策略

在多阶段决策过程中，如果决策变量$x_k(s_k)$已经确定，其中$k = 1, 2, \cdots, n$，那么函数序列$\{x_1(s_1),\ x_2(k_2), \cdots, x_n(k_n)\}$称为过程的一个策略。特别地，策略中的某一阶段开始到终点的决策集合序列称为原策略的子策略。

（6）指标函数和最优指标函数值

指标函数又称为目标函数，它是用于衡量所选策略优劣的数量指标。指标函数是定义在过程上的数量函数，通常用$V_{k,n}$表示，$V_{k,n} = V_{k,n}(s_k, x_k, s_{k+1}, x_{k+1}, \cdots, s_n, x_n)$，其中$k = 1, 2, \ldots, n$。指标函数由初始状态和策略决定，因此指标函数同时也是初始状态和策略的函数，即$V_{k,n} = V_{k,n}[s_k, P_{n,k}(s_k)]$。

指标函数$V_{k,n}$的最优值称为最优指标函数值，当k阶段的初始状态s_k给定时，最优指标函数值记为$f_k(s_k)$，$f_k(s_k) = \text{opt}\{V_{k,n}(s_k, x_k, s_{k+1}, x_{k+1}, \cdots, s_n, x_n)\}$，在实际问题中opt分别表示max或者min。

下面要给出运用动态规划的一个重要条件，它被称为“无后效性”原则。它的具体内容是将各阶段按照一定的次序排列好之后，对于某个给定的阶段状态，它以前各阶段的状态无法直接影响它未来的发展，而只能通过当前的这个状态。

因为动态规划是按顺序去求每阶段的解，如果一个问题有后效性，那么这样的次序便是不合理的。当然，可以通过重新划分阶段，重新选定状态，或者通过增加状态变量的个数等手段，来使问题满足无后效性这个条件。这个原则在实际应用中请读者务必注意。

动态规划没有固定的模式和统一的步骤，但通常情况下，运用动态规划的方法解决实

际问题时，可以遵循一个基本的路线来进行。为了帮助读者充分理解这一过程，这里配合一个求解最短路径的实例来进行说明。

前面曾经介绍过求最短路的算法：即狄克斯特拉算法和弗洛伊德算法。其中狄克斯特拉算法可用于计算图中某一点到各点的最短路径。弗洛伊德算法可用于计算图中所有顶点之间的最短路径。动态规划方法也可以用于求最短路径问题。

用动态规划方法求解最短路径问题时，要求所求问题具有明显的阶段，也就是说运用动态规划的方法解决实际问题的第一步就是将过程进行恰当的分段，这种分段应当能够便于把问题的过程转化成多阶段决策的过程。实际中有些问题不能直接划分出阶段。对于这种情况可以通过一定的转化方法将实际问题转化成阶段固定的标准模型。图 8-37 是最短路径标准模型。

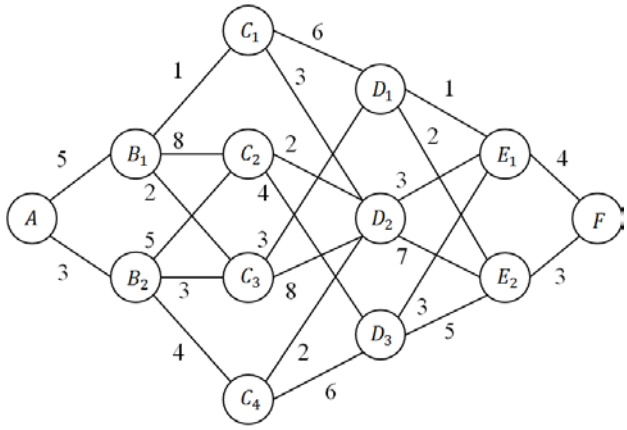


图 8-37　最短路径标准模型

下面以动态规划求解最短路径的标准模型为例，说明应用动态规划方法求解最短路径的方法，并通过这个例子来说明应用动态规划进行问题求解的基本步骤。图 8-37 是一个区域交通线路网，A是起始点，F是终点，图中的边表示道路，边上的权值表示道路的距离。请选择一条从A到F的路线，使得总距离最短。前面曾经提过，最短路径有一个重要的特性就是如果从起点到终点的最短路径经过点P，则由点P出发到达终点的路径也是所有可能的路径中最短的。鉴于这个特性，不妨从最后一段开始，从后向前，逆向推导最短路径，本例就采取这种解法。

显然，对于这个标准模型，问题被分为了 5 个阶段。即$A \to B$、$B \to C$、$C \to D$、$D \to E$、$E \to F$这 5 段。

接下来要选择状态变量s，使s既能反映过程演变的特征，又能满足无后效性，还能通过直接或间接的方法得知它在各阶段的变量值。无后效性，又称为马尔可夫性，它是指如果某段状态给定，则在这段以后的过程的发展只与当前的状态有关，而与过去的历史无关。之所以要求s满足该特性是因为如果s无法满足该特性，那么就会导致规划过程不能唯一确定下阶段的状态及其最优指标函数。从本例中可以看出，每个阶段的状态变量s分别是：

$$s_1 = \{A\}, \qquad s_2 = \{B_1, B_2\}, \qquad s_3 = \{C_1, C_2, C_3, C_4\}, \cdots$$

在确定状态变量s之后，需要确定决策变量及每阶段的允许决策集合$D_k(s_k)$。而在本例中，当$k = 5$时，由E_1到终点F的路线只有一条，即这条路线就是该阶段的决策变量。当确定了决策变量及每阶段的允许决策集合之后，需要根据实际问题的变化过程写出状态转移方程和目标函数，并据此进行问题的求解。E_1到终点F的路线只有一条，所以$f_5(E_1) = 4$；由E_2到终点F的路线也只有一条，故$f_5(E_2) = 3$。

继续求解，当$k = 4$时，易见出发点有D_1、D_2、D_3这 3 个。如果选择D_1为出发点，则有两条路线可供选择，一条通向E_1，另一条通向E_2，所以

$$f_4(D_1) = \min\{d_4(D_1, E_1) + f_5(E_1), d_4(D_1, E_2) + f_5(E_2)\} = \min\{5, 5\} = 5$$

此时最短路线有两条，一是$D_1 \to E_1 \to F$，对应决策为$x_4(D_1) = E_1$；以及$D_1 \to E_2 \to F$，相应的决策为$x_4(D_1) = E_2$。

同理，如果选择D_2为出发点，也有两条路线可供选择，一条通向E_1，一条通向E_2，所以有：

$$f_4(D_2) = \min\{d_4(D_2, E_1) + f_5(E_1), d_4(D_2, E_2) + f_5(E_2)\} = \min\{7, 10\} = 7$$

这表明其最短路径应该是$D_2 \to E_1 \to F$，决策为$x_4(D_2) = E_1$。

同理，如果选择D_3为出发点，也有两条路线可供选择，一条通向E_1，一条通向E_2，所以有：

$$f_4(D_3) = \min\{d_4(D_3, E_1) + f_5(E_1), d_4(D_3, E_2) + f_5(E_2)\} = \min\{7, 8\} = 7$$

这表明其最短路径应该是$D_3 \to E_1 \to F$，且$x_4(D_3) = E_1$。

当$k = 3$时，出发点有 4 个，即C_1、C_2、C_3、C_4。如果选择C_1为出发点，则有两条路线可供选择，一条通向D_1，另一条通向D_2，所以：

$$f_3(C_1) = \min\{d_3(C_1, D_1) + f_4(D_1), d_3(C_1, D_2) + f_4(D_2)\} = \min\{11, 10\} = 10$$

其最短路线是$C_1 \to D_2 \to E_1 \to F$，且$x_3(C_1) = D_2$。

同理，从C_2、C_3、C_4出发，则有：

$$f_3(C_2) = \min\{d_3(C_2, D_2) + f_4(D_2), d_3(C_2, D_3) + f_4(D_3)\} = \min\{9, 11\} = 9$$

其最短路线是$C_2 \to D_2 \to E_1 \to F$，且$x_3(C_2) = D_2$。

$$f_3(C_3) = \min\{d_3(C_3, D_1) + f_4(D_1), d_3(C_3, D_2) + f_4(D_2)\} = \min\{8, 13\} = 8$$

其最短路线是$C_3 \to D_1 \to E_1 \to F$或$C_3 \to D_1 \to E_2 \to F$，且$x_3(C_3) = D_1$。

$$f_3(C_4) = \min\{d_3(C_4, D_2) + f_4(D_2), d_3(C_4, D_3) + f_4(D_3)\} = \min\{9, 13\} = 9$$

其最短路线是$C_4 \to D_2 \to E_1 \to F$，且$x_3(C_4) = D_2$。

当$k = 2$时，出发点有两个，即B_1和B_2，则有：

$$f_2(B_1) = \min\{d_2(B_1, C_1) + f_3(C_1), d_2(B_1, C_2) + f_3(C_2), d_2(B_1, C_3) + f_3(C_3)\}$$

$$= \min\{11,17,10\} = 10$$

其最短路线是$B_1 \to C_3 \to D_1 \to E_1 \to F$或$B_1 \to C_3 \to D_1 \to E_2 \to F$，且$x_2(B_1) = C_3$。

$$f_2(B_2) = \min\{d_2(B_2, C_2) + f_3(C_2), d_2(B_2, C_3) + f_3(C_3), d_2(B_2, C_4) + f_3(C_4)\}$$

$$= \min\{14, 11, 13\} = 11$$

其最短路线是$B_2 \to C_3 \to D_1 \to E_1 \to F$或$B_2 \to C_3 \to D_1 \to E_2 \to F$，且$x_2(B_2) = C_3$。

当$k = 1$时，出发点只有一个A，则$f_1(A) = 14$，$x_1(A) = B_2$。

最终求得图 8-37 中A到F的最短路径是$A \to B_2 \to C_3 \to D_1 \to E_1 \to F$或$A \to B_2 \to C_3 \to D_1 \to E_2 \to F$, 相应的路径长度为14, 并得到最优决策策略为$x_1(A) = B_2, x_2(B_2) = C_3$, $x_3(C_3) = D_1$，$x_4(D_1) = E_1$或$x_4(D_1) = E_2$，$x_5(E_1) = F$或$x_5(E_2) = F$。

从以上这个例子中我们归纳得到运用动态规划进行问题求解所遵循的一般模式通常可分为如下几个阶段。

（1）划分阶段

在该阶段中需按照问题的时空特征，对问题进行划分，形成求解过程的若干个阶段。注意这若干个阶段一定要是有序的或者是可排序的，否则问题就是无法求解的。

（2）选择状态

在该阶段中，我们将问题发展到各个阶段时所处于的状态表示出来。注意状态的选择必须满足无后效性原则。

（3）决策并写出状态转移方程

状态转移是根据上一阶段的状态和决策来导出本阶段的状态。因此，一旦决策确定，那么状态转移方程也就随即可以得到。当然，通常也可以反过来做，即根据相邻两段的各状态之间的关系来确定决策。

（4）写出规划方程（包括边界条件）

经过上述诸多步骤后，阶段、状态、决策和状态转移都已被相继确定，那么最终得到规划方程和边界条件就非常容易了。

从上例中可以看出应用动态规划方法求解问题的关键在于正确地得到k阶段与$k+1$阶段之间的递推关系和边界条件。

通常k阶段与$k+1$阶段之间的递推关系可以表示为：

$$f_k(s_k) = \operatorname*{opt}_{x_k \in D_k(s_k)} \{V_k[s_k, x_k(s_k)] + f_{k+1}[x_k(s_k)]\}, \qquad k = n, n-1, \cdots, 2, 1$$

边界条件可以表示为$f_{n+1}(s_{n+1}) = 0$。

上述给出的是动态规划求解问题的一般模式，更具体地说，设计一个动态规划算法的一般步骤如下。

（1）分析最优解的性质，并刻画其结构特征。

（2）递归地定义最优值。

（3）以自底向上的方式或自顶向下的记忆化方法计算出最优值。

（4）根据计算最优值时得到的信息，构造一个最优解。

上述步骤（1）至步骤（3）是动态规划的基本步骤。只有当需要求出问题的一个最优解时，才必须执行步骤（4）。此时，在步骤（3）中计算最优值时，通常需记录更多信息，以便在步骤（4）中，根据所记录的信息，快速地构造出一个最优解。这一点在本节所给出的最短路径例子中体现得非常明显。

需要说明的是动态规划是一种问题求解思想，不是具体的算法，因此归纳出一个绝对的模式或框架是不可能的。前面得到的结论仅是在大多数情况下比较普遍和通用的设计思路。我们希望提供给读者的是一种思路而非一种束缚。希望读者不要太过拘泥于模式，不要太追求套路，这样做只会限制思维，扼杀灵感。尽管本书介绍动态规划的篇幅有限，但本节所述的内容可以作为动态规划算法的入门导引，相信有兴趣的读者在此基础之上参阅学习其他书籍中的相关内容将会事半功倍。

8.5　最小生成树

本章 8.1 节中曾经给出了连通的概念。图的连通性是现实中存在的具有重要研究价值的问题。对于一个无向非连通图而言，可以定义其中的每个连通部分都是它的一个极大连通子图。相应地，对于一个无向连通图，也可以定义其中任何一个使得所有顶点都相互连通的一个子图为它的一个极小连通子图。如何在这些极小连通子图中寻找一个“规模最小的”，就是本节要研究的最小生成树问题。

8.5.1　最小生成树的定义

假设存在一个连通的无向图，它的一个生成树就是该图的一个子图，且这个子图是一个将所有顶点都连接起来的一棵树。一个图可能存在很多不同的生成树。例如，图 8-38 中的 4 个子图就都是图 8-2 中的连通图的若干个生成树。

对于带权图，每条边的权值都是一个用来表示该边消耗和代价的数字，如果通过计算生成树中边的权值之和，就可以求得一个权值和最小的生成树，它就是所谓的最小生成树（或称最小权值生成树）。图 8-38 右下角的一棵生成树就是其原图的最小生成树。

更普遍的情况下，任何无向图中都存在最小生成森林，最小生成森林就是原图中所有极大连通子图的最小生成树的并集。通过研究连通图中的最小生成树可以很容易推广到最小生成森林的情况。

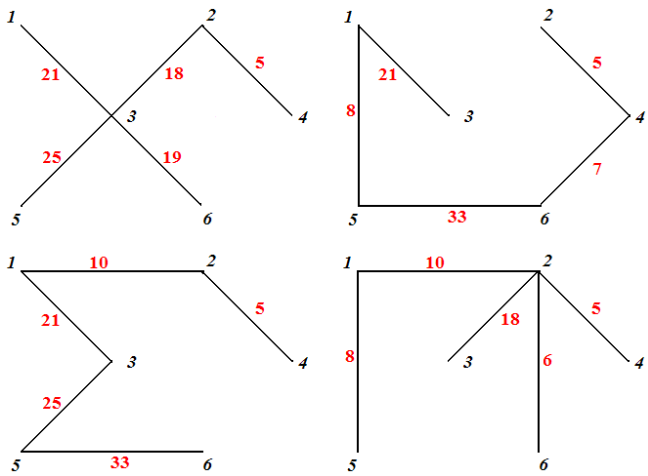


图 8-38　生成树

求解图的最小生成树往往对应现实生活中的某些最优化问题。例如，一个电视电缆公司希望在邻近的一些地点间铺设线缆。线缆必须沿着一定的路径被埋进地里，并将所有需要服务的地点连接起来。可以使用一个图来表示所有可能路径和需要提供服务的地点。电视电缆公司希望铺设线缆的费用最低，但是如何找到一条最便宜的铺设线路呢？因为所有可能的途径中的某些部分或许铺设费用非常昂贵，因为它们比较长，或者由于地理原因而需要把电缆埋得很深。这些路径可以用具有较大权值的边来表示。该图的一个极小生成树就是一个连接所有地点且没有圈的子树。也就是说极小生成树是所有可能路径的一个子集，这些路径的特点是没有圈但连通每一个地点。那么最小生成树就是这些极小生成树中的那个建设费用最低的。

捷克数学家奥托卡·布鲁夫卡（Otakar Boruvka）在 1926 年发明了第一个用来解决最小生成树问题的算法——布鲁夫卡算法。布鲁夫卡当时的目的是利用该算法为摩拉维亚地区设计一个有效的电网覆盖方案。目前使用较为广泛的两个算法分别是克鲁斯卡尔算法和普里姆算法，它们也是本节需要重点介绍的方法。以上的 3 个方法都是使用贪婪策略设计的，另一个不是经常被用到的应用贪婪策略设计的算法是逆向删除算法，它相当于克鲁斯卡尔算法的反过程。由于逆向删除算法并不常用，本节将不会对其进行介绍。

在设计更加高效的最小生成树算法方面，研究并未就此停止。其中最具代表性的成果是由计算机科学家伯纳德·沙泽勒（Bernard Chazelle）于 1997 年提出的基于 Soft 堆的最小生成树算法，其时间复杂度仅为$O(m\alpha \log \alpha)$，其中$\alpha = \alpha(m,n)$表示阿克曼函数的反函数，n表示节点的数量，相应地m表示边的数量。沙泽勒的工作将最小生成树算法又推向了一个新的高度。后来，他又在 2000 年改进了该算法，将算法的时间复杂度提高到$O(m\alpha(m,n))$。这几乎是目前基于非并行方式下构建的性能最好的最小生成树算法。鉴于该算法所涉及的数据结构比较复杂，本书不对其进行讨论，有兴趣的读者可以查阅相关文献以了解更多。

8.5.2　克鲁斯卡尔算法

算法

求解最小生成树的克鲁斯卡尔算法。

克鲁斯卡尔算法由美国计算机科学家约瑟夫·克鲁斯卡尔（Joseph Kruskal）于 1956 年提出，其核心思想是每次添加权尽可能小的边，同时使新的图无环，直至得到生成树为止。因此，该方法也被形象地称为“最小边加入法”或“避圈法”。

克鲁斯卡尔算法的步骤可以简单描述如下，假设有带权图$G(V,E)$，首先将图G中的边按权值排序，并将其中权值最小的边选出并标记，如果具有最小权值的边有多条，则可以随机从中选择一条。然后从未标记的边中选出权值最小的一条，并检查该边是否与已经标记的边构成圈。如果是，则废弃它重新选择一条权值最小的边并做检查，直至找到一条与已标记的边不会构成圈的权值最小的边，然后将其标记，即该边成为最小生成树的一部分。如此往复下去，直到使图中所有顶点都相互连通为止。对于具有n个顶点的连通图，其最小生成树应当由$n-1$条边构成。

下面举例说明克鲁斯卡尔算法的工作过程。对如图 8-2 所示的连通图求其最小生成树。如图 8-39 所示，算法首先将图中的所有边擦除，仅保留图中的顶点。

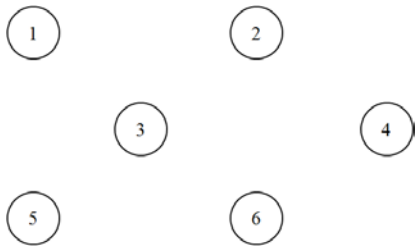


图 8-39　克鲁斯卡尔算法示例（1）

算法扫描图的边，并选出其中的权值最小的边(2,4)，将其标记，如图 8-40 所示。

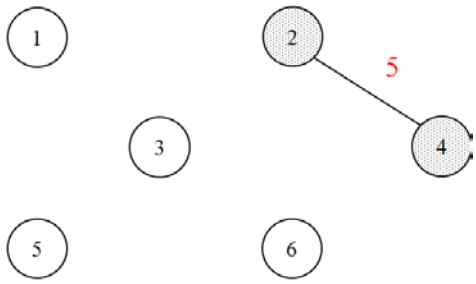


图 8-40　克鲁斯卡尔算法示例（2）

算法继续进行，从未标记的边中选出权值最小的(2,6)，因为该边不会和已标记的边构成圈，所以对其进行标记，使它成为最小生成树的一部分，这个过程如图 8-41 所示。

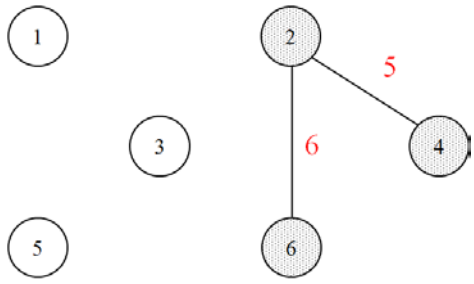


图 8-41　克鲁斯卡尔算法示例（3）

算法继续从未标记的边中查找具有最小权值的边，结果发现了边(6,4)，但是算法不能标记它，因为它会和已标记的边构成圈。于是算法废弃它，继续寻找适合的边，发现了边(1,5)。易见该边符合条件，于是标记它，如图 8-42 所示。

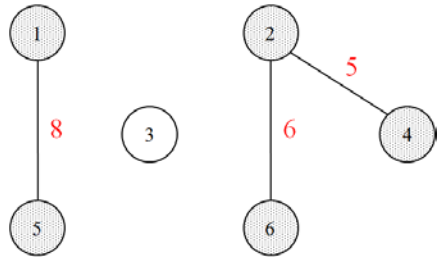


图 8-42　克鲁斯卡尔算法示例（4）

算法重复上述过程，完成剩余工作，如图 8-43 和图 8-44 所示，最终求得原图的最小生成树。

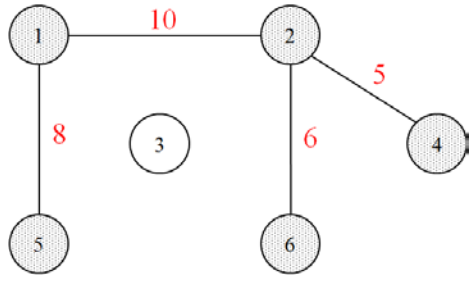


图 8-43　克鲁斯卡尔算法示例（5）

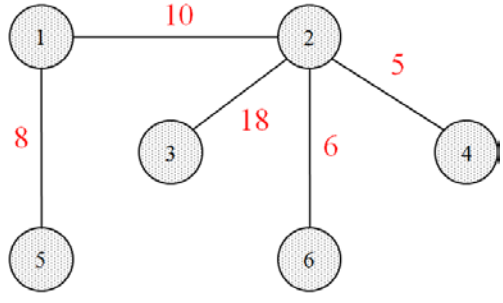


图 8-44　克鲁斯卡尔算法示例（6）

经典问题求解

道路修建问题，这也是一道常见的 ACM 竞赛练习题。

最后我们用克鲁斯卡尔算法来解决一个实际问题——道路修建问题。现有N个村子，这些村子的编号是从 1 到N。你要修建一些路，这些路应把这些村子两两连接起来。如果A村与B村相连，要么A村与B村之间有一条路，要么A村与C村之间有一条路，并且B村与C村之间也有一条路。我们已知村子与村子之间已经修建了一些路。你的任务是修建剩下的路以使得所有的村子都连接上并且路的总长度最小。（题目来源：PKU Monthly, KICC）

输入：第 1 行包含一个整数N，其中N是村子的数量，$3 \leqslant N \leqslant 100$。接下来有$N$行，这$N$行中的第$i$行包含$N$个整数，这$N$个整数中的第$j$个代表了村与村之间的距离（这个距离应在区间[1, 1000]内），即i村与j村之间的距离。接下来是一个整数Q，其中满足$0 \leqslant Q \leqslant N \times (N+1)/2$。然后是$Q$行数据，每一行包含两个整数$a$与$b$，$1 \leqslant a < b \leqslant N$，这意味着村$a$与村$b$之间的路已修建。下面给出一段输入示例。

```
3
0 990 692
990 0 179
692 179 0
1
1 2
```

输出：你的程序应在一行中输出一个整数，这个数代表了要把这些村子连接起来所需的最短总路程。例如，若是按照上述给出的示例进行输入，将得到输出 179。

显然这是一个在图中求最小生成树的问题。示例代码按照克鲁斯卡尔算法在图中求解最小生成树并成功解决了该问题。下面给出程序实现的源代码。

```
#include "stdafx.h"
#include "stdio.h"
#include "stdlib.h"
#include "iostream"

using namespace std;

#define N 110

typedef struct{
    int prior;int next;
    int dis;
}Node;

Node node[N*N];
int father[N];
int find(int);
int union_(int ,int);
int MST_Kruskal();
int n,ans=0;
int cmp(const void *a,const void *b) {
    Node *x=(Node*)(a);Node *y=(Node*)(b);
    return (x->dis-y->dis);
}
```

```
int _tmain(int argc, _TCHAR* argv[])
{
    int m=0;
    int i,j,a,b,t=0;
    scanf("%d",&n);
    for(i=1;i<n+1;i++)
        for(j=1;j<n+1;j++) {
            node[t].prior=i;
            node[t].next=j;
            scanf("%d",&node[t++].dis);

        }
    scanf("%d",&m);
    for(i=1;i<N;i++)
        father[i]=-1;
    for(i=0;i<m;i++) {
        scanf("%d%d",&a,&b);
        union_(a,b);
    }
    qsort(node,n*n,sizeof(Node),cmp);
    MST_Kruskal();

    cout<<ans<<endl;
    system("PAUSE");
    return 0;
}

int MST_Kruskal() {
    int i=0,u,v;
    for(i=0;i<n*n;i++) {
            u=node[i].prior;
            v=node[i].next;
            if(find(u)!=find(v)) {
                    union_(u,v);
                    ans+=node[i].dis;
            }
    }
    return 0;
}

int find(int a ){
    int i,temp;
    for(i=a;father[i]>0;i=father[i]);
    while(a!=i) {
        temp=father[a];
        father[a]=i;
        a=temp;
    }
    return i;
}

int union_(int a,int b) {
    int temp,temp1=find(a),temp2=find(b);
    if(temp1==temp2) return 0;
    temp=father[temp1]+father[temp2];
       father[temp2]=temp1;
       father[temp1]=temp;
       return 1;
}
```

请读者自行编译并执行程序后，观察输出结果。

8.5.3　普里姆算法

算法
求解最小生成树的普里姆算法。

普里姆算法最初由捷克数学家沃伊捷赫·亚尔尼克（Vojtěch Jarník）于 1930 年提出，并在 1957 年由美国计算机科学家罗伯特·普里姆（Robert Prim）再次独立提出。

假设存在连通带权图$G(V,E)$，其最小生成树为T。那么使用普里姆算法生成T的过程如下：首先从图中随意选择一点$u_0 \in V$作为起始点，并将其标记后加入集合U中，即此时应该有$U=\{u_0\}$。然后算法重复执行以下操作：在所有$u \in U$，$v \in (V-U)$的边$(u,v) \in E$中找一条代价最小的边(u_0,v_0)并入集合T，同时将v_0并入U，直至$U=V$为止。这时，T中必有$n-1$条边，这时T就是G的最小生成树。

为了深化读者对于该算法的理解，下面举例说明普里姆算法是如何工作的，这里同样对如图 8-2 所示的连通图求其最小生成树。算法首先选择任意顶点作为起始点，这里选择顶点4作为起始点。将顶点4放入集合U中，如图 8-45 所示，将其着色。然后算法在未标记的顶点中选择一个顶点v_0使得边$(4,v_0)$的权值最小。显然可能选择的点将构成的边有(4,2)和(4,6)。又知Weight(4,2) = 5，Weight(4,6) = 7。因此，选择顶点2，而非顶点 5。标记顶点2并将其加入到集合U中。

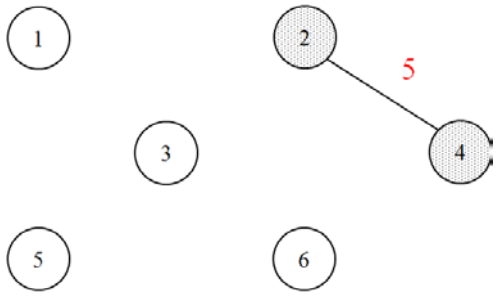


图 8-45　普里姆算法示例（1）

算法继续进行。现在算法试图从未标记的点集$U-V$中找到一点v_1使得边(u,v_1)的权值最小，其中$u \in U=\{2,4\}$。显然可能选择的点将构成的边有(2,1)、(2,3)、(2,6)、(4,6)。其中权值最小的边是(2,6)，所以标记顶点6并将其加入到集合U中，该过程如图 8-46 所示。

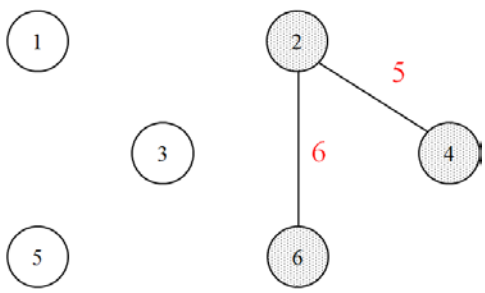


图 8-46　普里姆算法示例（2）

此时，算法试图从未标记的点集$U-V$中找到一点v_2使得边(u,v_2)的权值最小，其中$u\in U=\{2,4,6\}$。显然可能选择的点将构成的边有(2,1)、(2,3)、(6,3)、(6,5)。易见，其中权值最小的边是(2,1)，所以标记顶点1并将其加入到集合U中，该过程如图 8-47 所示。

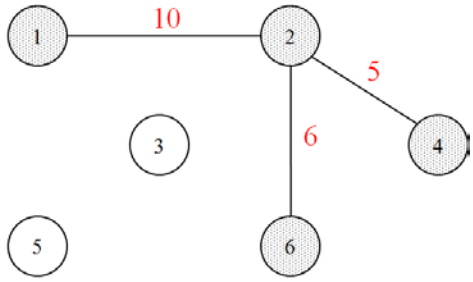


图 8-47　普里姆算法示例（3）

算法如此继续下去，如图 8-48 和图 8-49 所示，直到$U=\{1,2,3,4,5,6\}=V$，即图中所有的点都已被标记，此时算法终止，最小生成树被获得。

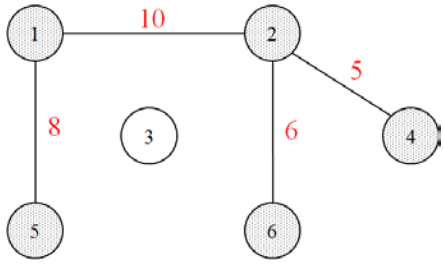


图 8-48　普里姆算法示例（4）

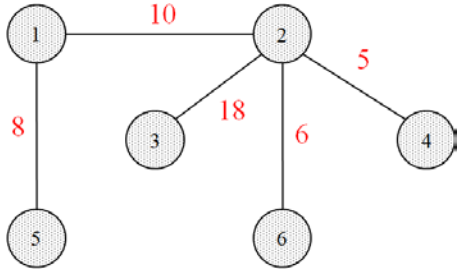


图 8-49　普里姆算法示例（5）

最后需要指出，克鲁斯卡尔算法的时间复杂度为$O(e\log e)$，其中e是图中边的数目。普里姆算法的时间复杂度为$O(n^2)$。克鲁斯卡尔算法适用于稀疏图，而普里姆算法适用于稠密图。两个算法的弱点都在于不能根据点数、边数及边的分布动态调整自身的性能，因此已有学者从该角度提出了优化方案，读者也可以尝试思考这两个算法的优化方案。

下面给出使用普里姆算法求解最小生成树的示例代码。该例子使用邻接矩阵作为图的存储形式。

```
#include "stdafx.h"
#include "iostream"

using namespace std;

#define MaxNumVer 10
#define MaxWeight 100

int _tmain(int argc, _TCHAR* argv[])
{
    cout<<"请注意:"<<endl;
    cout<<"\t 本程序默认的最大顶点数为 10"<<endl;
    cout<<"\t 边权值的取值范围是介于 0 到 99 之间的"<<endl<<endl;

    int G[MaxNumVer][MaxNumVer],in[MaxNumVer]={0},path[MaxNumVer][2];
    int i,j,k,min=MaxWeight;
    int v1,v2,num,tmpValue,status=0,start=0;

    //输入图节点的数目
    while(true)
    {
        cout<<"请输入图中顶点的数目:"<<endl;
        cin>>num;
        if(num<=MaxNumVer&&num>0)
            break;
        cout<<"错误!请确认后重新输入."<<endl<<endl;
    }

    //创建图并使用邻接矩阵 G 来存储图
    for(j=0;j<num;j++)
        for(k=0;k<num;k++)
        {
            if(j==k)
                G[j][k]=MaxWeight;
            else if(j<k)
            {
                while(true)
                {
                    cout<<"请输入边("<<j+1<<","<<k+1<<")的权值,
                            如果改变不存在请输入-1:"<<endl;
                    cin>>tmpValue;
                    if(tmpValue>=-1&&tmpValue<MaxWeight)break;
                    cout<<endl<<"输入无效!请确认后重新输入."<<endl;
                }

                if(tmpValue==-1)
                    tmpValue=MaxWeight;

                G[j][k]=G[k][j]=tmpValue;//矩阵是对称的
            }
        }

    for(j=0;j<num;j++)
    {
        status=0;
        for(k=0;k<num;k++)
            if(G[j][k]<MaxWeight)
            {
```

```
                status=1;
                break;
            }
        if(status==0)
            break;
    }

    //如果图是非连通的则退出程序
    if(!status){
        cout<<"无法处理，因为该图是非连通的!"<<endl;
        return -1;
    }

    //选择算法的起始点
    while(true){
        cout<<"请选择普里姆算法的起始点:"<<endl;
        cin>>start;

        if(start>0 && start<=num)
            break;

        cout<<"错误!请确认后重新输入."<<endl<<endl;
    }

    //普里姆算法求最小生成树
    in[start-1]=1; //标记
    for(i=0;i<num-1;i++)
    {
        for(j=0;j<num;j++)
            for(k=0;k<num;k++)
                if(G[j][k]<min&&in[j]&&(!in[k]))
                {
                    v1=j;
                    v2=k;
                    min=G[j][k];
                }

        if(!in[v2])
        {
            path[i][0]=v1;
            path[i][1]=v2;

            //标记
            in[v1]=1;
            in[v2]=1;

            min=MaxWeight;
        }
    }

    //输出结果
    cout<<"最小生成树如下:"<<endl;
    for(i=0;i<num-1;i++)
        cout<<"("<< path[i][0]+1 <<", "<< path[i][1]+1<< ")" <<endl;

    system("PAUSE");
    return 0;
}
```

请读者完成编码后，运行程序并观察程序执行结果。

本章参考文献

[1] Dijkstra, E. W. A note on two problems in connexion with graphs. Numerische Mathematik, 1: 269~271, 1959.

[2] Floyd, R. W. Algorithm 97: Shortest Path. Communications of the ACM, Vol. 5, No. 6: 345. 1962.

[3] Prim, R. C. Shortest connection networks and some generalizations. Bell System Technical Journal, Vol. 36, No. 6: 1389~1401, 1957.

[4] Kruskal, J. B. On the shortest spanning subtree of a graph and the traveling salesman problem. Proceedings of the American Mathematical Society, 7: 48~50, 1956.

[5] Morin, T. L. Monotonicity and the principle of optimality. Journal of Mathematical Analysis and Applications, Vol. 88, No. 2: 665~674, 1982.

[6] Chazelle, B. A faster deterministic algorithm for minimum spanning trees. Proceedings of the 38th Annual Symposium on Foundations of Computer Science, 38: 22~31, 1997.

[7] Chazelle, B. A minimum spanning tree algorithm with inverse-Ackermann type complexity. Journal of the ACM, Vol. 47, No. 6: 1028 ~1047, 2000.

[8] 庞素超, 陈实. 用动态规划方法求解最短路径问题. 大庆石油学院学报. 2007 年, 第 31 卷, 第 3 期.

[9] 方世昌. 离散数学（第 2 版）. 西安：西安电子科技大学出版社, 2002.

[10] 赵静, 但琦. 数学建模与数学实验. 北京：高等教育出版社, 2000.

[11] 秦裕瑗. Bellman 最优性原理——论动态规划 (I). 应用数学. 1994 年, 第 7 卷, 第 3 期.

第 9 章

树形搜索结构——做一名出色的园艺师

搜索也称查找，它是计算机中最重要也最常用的操作之一。它的重要性不亚于甚至高于后面章节中将要讲述的排序操作。那么何谓搜索呢？基于数据表和关键码这两个概念可以很容易地给出搜索的定义。搜索就是从数据表中寻找与关键码相匹配的数据项的过程。当数据量很大时，搜索操作对于任何一台计算机来说都是非常繁重的，这个问题长久以来始终是学者们关注的热点。除了不断研发更加先进的计算机硬件设备以外，设计更加高效的搜索方法也是解决该问题的一个重要途径。通常根据搜索方法所采用的不同的数据结构，可以将它们分为 3 类，即基于线性结构的搜索方法、基于哈希的搜索方法和基于树的搜索方法。本章将向读者介绍有关树形搜索的一些内容。

9.1　二叉搜索树

如果将大量数据存储在线性表中，那么计算机执行搜索任务的负担是非常重的。使用二叉搜索树是提高搜索效率的一个途径。二叉搜索树是基于折半查找思想而设计的一种简单的树形数据结构。在最好的情况下，二叉搜索树进行操作的复杂度仅为$O(\log N)$。

小提示

二叉搜索树和 AVL 树属于基础内容，应该掌握。后面的红黑树属于高级内容，供有兴趣的读者参阅学习。

9.1.1　二叉搜索树的概念

二叉搜索树是一种特殊的二叉树，它将数据表中每个数据项的关键码存储在树中对应的每个节点里，并且对于树中任意节点X的左右子树应当满足这样一个条件，即X的左子树中每个节点的关键码均应该小于节点X的关键码，而X的右子树中每个节点的关键码均应该大于节点X的关键码。满足上述条件的二叉树就是一棵二叉搜索树。如图 9-1 所示为一个二叉搜索树的例子。

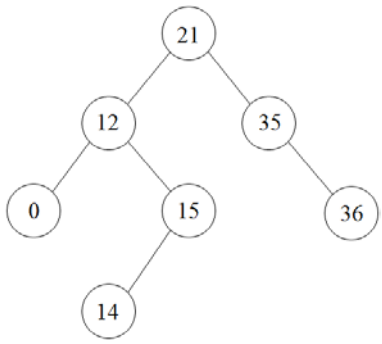


图 9-1　二叉搜索树

由二叉搜索树的定义可以知道，二叉树中的节点是处于有序状态的，但是其定义也告诉我们一个信息，那就是二叉搜索树节点中所存储的关键码是不允许重复的。这是一个基本的原则，如果违背它那么二叉树是不可能形成二叉搜索树的。但是现实世界中存在大量的重复情况，如果绝对要求避免重复那么二叉搜索树的局限性就变得很大。幸运的是，事实上，人们设计了很多辅助方法来帮助解决二叉搜索树中的关键码重复问题。这里介绍一种解决该问题的简单方法。使用一个辅助线性表是很好的想法，这个线性表用于存储原数据表中重复数据项的关键码及该数据项重复出现的次数。这样就很容易地解决了数据项关键码的重复问题，当然这不是唯一的方法，读者也可以尝试还有没有其他的好办法。注意，本节所讲的二叉搜索树是指标准型二叉搜索树，即不考虑存在重复元素的情况。

求职面经点拨

请试回答附录题库中的第 28 题。

9.1.2　二叉搜索树的操作

由于二叉搜索树的特殊属性和结构，在其中执行一些特殊操作是非常容易的。作为一种用来进行查找操作的结构，二叉搜索树中最重要的操作当然就是查找了。在二叉搜索树中进行查找操作是非常方便的。我们只需要从根节点开始重复地进行比较并根据比较结果来决定分支方向即可。例如，这里需要在如图 9-1 所示的二叉搜索树中搜索关键码为 14 的数据项。查找操作从该二叉搜索树的根开始并将 14 与当前节点中的关键码进行比较，由于 14 小于 21 所以分支转向该节点的左子树。再将 14 与当前节点中的关键码即 12 进行比较，由于 14 大于 12 所以分支转向该节点的右子树。再将 14 与当前节点中的关键码即 15 进行比较，由于 14 小于 15 所以分支转向该节点的左子树。最后查找成功。整个过程如图 9-2 所示。

如果待搜索的数据项的关键码是 13，那么其过程大体与搜索 14 的过程相似。但是当将 13 与 14 进行比较的时候，由于 13 小于 14，所以分支将转向存储 14 的节点的左子树。而该节点的左子树为空，即该节点的 LeftChild 指针为 NULL。这时搜索过程结束，搜索结果为数据表中不存在关键码为 13 的数据项。

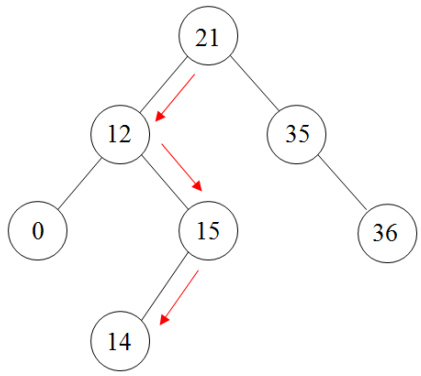


图 9-2　在二叉搜索树中执行查找操作

在二叉搜索树中搜索数据表中关键码最小或者最大的数据项也非常容易。如果要执行 FindMin（搜索关键码最小的数据项）操作，只需从根节点开始自顶向下重复地向每个节点的左分支前进，直至找到该二叉搜索树的一个叶子节点为止，该节点所包含的项目就是数据表中关键码最小的数据项。如果要执行 FindMax（搜索关键码最大的数据项）操作，则只需从根节点开始自顶向下重复地向每个节点的右分支前进，直至找到该二叉搜索树的一个叶子节点为止，该节点所包含的项目就是数据表中关键码最大的数据项。

如何向一个二叉搜索树中插入节点呢？这个过程与查找操作非常类似，不同的是我们仅仅是利用查找操作来确定插入节点的适当位置。例如，这里要向如图 9-1 所示的二叉搜索树中插入一个含有关键码 13 的节点。于是从该二叉搜索树的根开始并将 13 与当前节点中的关键码进行比较，由于 13 小于 21 所以分支转向该节点的左子树。再将 13 与当前节点中的关键码即 12 进行比较，由于 13 大于 12 所以分支转向该节点的右子树。以此类推，最后将 13 与节点 14 进行比较，由于 13 小于 14，所以分支将转向存储 14 的节点的左子树。而该节点的左子树为空，那么就将节点 14 的 LeftChild 指针指向节点 13，这样就完成了一个节点的插入。若是要插入一个关键码为 16 的节点，其方法也是如此，具体过程这里不再赘述。图 9-3 演示了上述操作的结果。

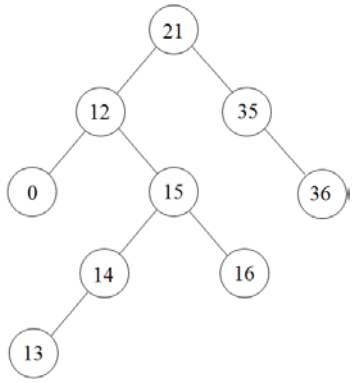


图 9-3　在二叉搜索树中执行插入操作

从一个二叉搜索树中删除一个节点相比之下要比前面的操作复杂得多。在从一棵二叉搜索树中删除一个节点之前必须先找到该节点所在的位置，关于二叉搜索树中查找操作前面已经讲过，这里不再重复。当确定了要删除之节点的位置时，就可执行删除操作。但是删除操作往往会引起很多后果，由其操作所引发的问题才是需要我们认真考虑的。这些问题主要来自两个方面，一方面删除树中的一个节点必然会导致原本连通的树结构发生阻隔，因此这时就需要恰当地将分开的部分连在一起。其次，当二叉搜索树中的某个节点被删除后，树中节点的有序关系可能被破坏，因此这时也需要采用恰当的办法来进行修补。

具体而言，在删除一个二叉搜索树节点时，需要考察该节点的子节点数目是多少。当该节点的子节点数目为 0 时，即该节点是一个叶子节点时，只需将其父节点相应的指针置为空后再将节点释放。如果该节点的子节点数目为 1，那么这时就需要将其父节点相应的指针改为指向待删除节点的子节点，然后再将该节点释放。例如，这里要删除如图 9-1 所示的二叉搜索树中的节点 15。其删除过程如图 9-4 所示。将节点 12 中本来指向节点 15 的指针（即 RightChild 指针）指向节点 14，然后将节点 15 释放，这样就完成了该节点的删除操作。

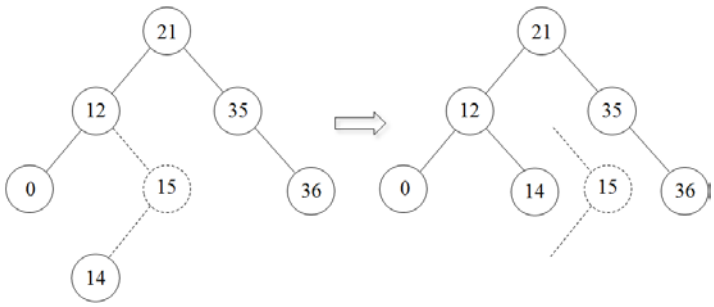


图 9-4　在二叉搜索树中执行删除操作（情况 1）

当一个二叉搜索树中的待删除节点具有两个子节点时，情况要更复杂些。因为我们不得不寻找新的合适节点来填充被删除节点的位置但又不能影响二叉搜索树中节点的有序性。这时所采取的办法是从该节点的右子树中查找具有最小关键码的节点，然后用该节点来替换被删除的节点并释放待删除节点。因为替换节点是被删除节点右子树中具有最小关键码的节点，所以它必然不可能是除了被删除节点以外其他节点的右子节点，且它也不可能有左子节点。所以替换节点只能是一个叶子节点或者只有一个子节点（即右子节点）。因为被删除节点的右子树中具有最小关键码的节点被用来替换已删除的节点了，所以这就相当于是将置换节点从树中原来的位置删除了（替换位置处的节点被看作是一个新插入的节点）。当置换节点是一个叶子节点时，无须进行其他操作。但当它具有一个子节点时，就需要按照前面所讲的方法将其父节点相应的指针改为指向它的子节点。

下面举一个简单的例子来说明当待删除节点具有两个子节点时二叉搜索树中的删除操作是如何进行的。如图 9-5 所示为从图 9-1 的二叉搜索树中删除节点 12 的具体过程。

因为节点 12 具有两个子节点，所以我们在其右子树中查找具有最小关键码的节点，查找结果为节点 14。于是用节点 14 来替换节点 12，再将节点 12 释放。因为节点 14 原来是一个叶子节点，所以属于节点删除操作中的第 1 种情况，因此不需要再做其他处理了。

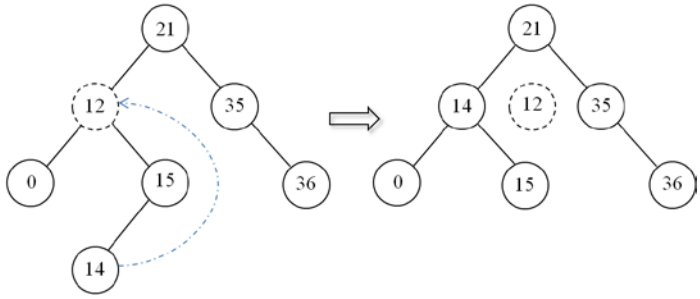


图 9-5　在二叉搜索树中执行删除操作（情况 2）

9.1.3　二叉搜索树的实现

本书前面曾经给出过二叉树实现的 C++代码，有了这样的基础之后，再来实现二叉搜索树就显得容易多了。下面首先给出二叉搜索树节点类的类定义。注意，其中所给出的代码仅仅能满足实现其功能的最低要求。我们略去了类中的其他成员函数，这些被略去的成员函数大部分与前面实现的普通二叉树中的成员方法相同，因此这里不再赘言。由于这个类仅仅是一个简单的实现，因此成员函数并不完整，为了不影响其功能，这里将其中的所有数据成员和成员函数声明为公有。此外，二叉搜索树类被声明为节点类的友元类。

```
class BinarySearchTree;

class BinaryNode
{
public:

    int element;
    BinaryNode * leftChild;
    BinaryNode * rightChild;

    BinaryNode(int theElement, BinaryNode * lt, BinaryNode * rt)
        :element(theElement), leftChild(lt), rightChild(rt) {};

    friend class BinarySearchTree;

    //其他成员函数略……
};
```

基于上面的节点类，下面我们给出二叉搜索树类的类定义。同样，这里仅仅给出了其中比较重要的成员函数声明，而略去了类中的其他成员函数。与 9.1.2 节所讲的内容对应，这里给出的成员函数包括：

- 查找函数 find。

- 查找最大值函数 findMax。
- 查找最小值函数 findMin。
- 插入节点函数 insert。
- 删除节点函数 remove。
- 删除最小值节点函数 removeMin。

```
class BinarySearchTree
{
public:
    BinaryNode * root;
    BinarySearchTree(BinaryNode * rt){root = rt;}

    BinaryNode * findMax(BinaryNode * t)const;
    BinaryNode * findMin(BinaryNode * t)const;
    BinaryNode * find(int x, BinaryNode * t)const;

    void insert(int x, BinaryNode * & t);
    void remove(int x, BinaryNode * & t);
    void removeMin(BinaryNode * & t);

    //其他成员函数略……
};
```

下面给出上述类定义中各成员函数的实现方法。

```
BinaryNode * BinarySearchTree::find(int x, BinaryNode * t)const
{
    while(t!=NULL)
        if(x < t->element)
            t = t->leftChild;
        else if(x > t->element)
            t = t->rightChild;
        else
            return t;

        return NULL;
}

BinaryNode * BinarySearchTree::findMax(BinaryNode * t)const
{
    if(t!= NULL)
        while(t->rightChild != NULL)
            t = t->rightChild;
    return t;
}

BinaryNode * BinarySearchTree::findMin(BinaryNode * t)const
{
    if(t!= NULL)
        while(t->leftChild != NULL)
            t = t->leftChild;
    return t;
}

void BinarySearchTree::insert(int x, BinaryNode * & t)
{
```

```
    if( t == NULL )
        t = new BinaryNode(x, NULL, NULL);
    else if ( x < t->element )
        insert(x, t->leftChild);
    else if ( x > t->element )
        insert(x, t->rightChild);
    else
    {
        cout<<"ERROR!!! Duplicate Element."<<endl;
        return;
    }
}

void BinarySearchTree::remove(int x,  BinaryNode * & t)
{
    if(t == NULL)
    {
        cout<<"This node does not exist!!!"<<endl;
        return;
    }
    if(x < t->element)
        remove(x, t->leftChild);
    else if(x > t->element)
        remove(x, t->rightChild);
    else if(t->leftChild != NULL && t->rightChild != NULL)
    {
        t->element = findMin(t->rightChild)->element;
        removeMin(t->rightChild);
    }
    else
    {
        BinaryNode * tmp = t;
        t = (t->leftChild != NULL) ? t->leftChild : t->rightChild;
        delete tmp;
    }
}

void BinarySearchTree::removeMin(BinaryNode * & t)
{
    if(t == NULL)
    {
        cout<<"Empty Tree!!!"<<endl;
        return;
    }
    else if(t->leftChild != NULL)
    {
        removeMin(t->leftChild);
    }
    else if(t->leftChild == NULL)
    {
        BinaryNode * tmp = t;
        t = t->rightChild;
        delete tmp;
    }
}
```

基于 9.1.2 节的学习，理解这几个函数的实现应当并不困难。其中，节点插入函数 insert 的实现使用了递归。节点删除函数 remove 也使用了递归，但 remove 函数相比之下可能更难理解一些。当程序判断得到待删除节点具有两个子节点时，它使用其右子树中

关键码最小的节点来替换该节点后再调用 removeMin 函数来将右子树中的原最小节点删除。

最后建议读者将上述类的定义和实现改写成模板形式，这样可以使该类更具通用性，同时可以帮助读者巩固对于 C++模板机制的熟悉和了解。完成改写后请读者再写一个主函数来测试这个类中的成员方法。

9.1.4　二叉搜索树的分析

二叉搜索树中的各主要操作（如 insert、find 和 remove）的时空消耗都与操作过程中被访问的节点数目相关。因此，可以使用成功搜索到某节点时探寻过程的深度来衡量该算法的复杂度。

如图 9-6 所示的一棵二叉搜索树共有 15 个节点，该树的特点在于它是一棵呈现平衡性质的二叉搜索树，因为其中任意节点的左子树和右子树都是等高的。对于这样的一棵平衡二叉搜索树而言，访问其中任何一个节点的代价最多不过是探寻深度为 4，而且树中部分节点的访问代价相比之下还要更低。这是一种非常理想的情况，在这种情况下，二叉搜索树中主要操作的复杂度仅为对数级。

求职面经点拨

请试回答附录题库中的第 29 题。

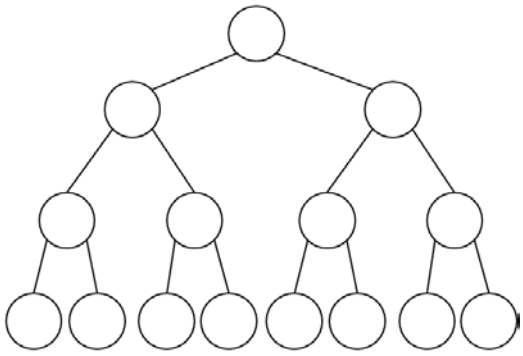


图 9-6　平衡树

不幸的是我们无法保证现实中的二叉搜索树都是平衡的，这种特殊情况往往很少，更多的情况是树中各节点分布呈不对称或不规则状态。一个极端的例子是前面曾经提过的“ 单支树 ”（请参看本书第 7 章）。对于一个具有 15 个节点的单支树而言，访问其中任何一个节点的代价最大探寻深度将变为 15。也就是说在这种情况下，搜索效率最差，时间复杂度将为$O(n)$，其中n是树中的节点数目。

当二叉搜索树平衡的时候，搜索效率最高，其时间复杂度为$O(\log n)$。当二叉搜索树是单支树时，其搜索效率最低，时间复杂度为$O(n)$。那么平均情况下，二叉搜索树的操作时间复杂度应当是多少呢？学术界暂时还不能给出这个问题的准确答案。不过实践所得到的结论是二叉搜索树操作的平均效率要比其最好情况下的效率差大约 38%，也就是二叉搜索树操作的平均时间复杂度是接近对数级的。这个结论目前还无法得到理论上的证明。但我们可以从二叉树内部路径长度和外部路径长度的角度来考察二叉

搜索树的搜索操作复杂度，这两个概念可以说是对其操作效率的一个侧面反映。其中，内部路径长度被用来衡量一个成功搜索的时间代价，而外部路径长度则可以用来衡量一个失败搜索的时间代价。

首先给出二叉树内部路径长度（IPL，Internal Path Length）的定义。所谓内部路径长度就是指一个二叉树内部所有节点的深度和。下面来讨论在一般情况下一棵二叉搜索树的内部路径长度平均应该是多少。此处设$D(N)$表示具有N个节点的一棵二叉搜索树的内部路径平均长度，易得$D(1) = 0$。如果N不等于0，那么这棵具有N个节点的二叉搜索树可以认为是由 3 部分组成的，即根节点、根节点的左子树和根节点的右子树。如果根节点的左子树中含有i个节点，那么根节点的右子树中就会含有$(N - i - 1)$个节点，其中$0 \leqslant i < N$（这里假设i在$[0, N]$范围内是等概率分布的）。对于一个给定的i，$D(i)$就是相应根节点的左子树的平均内部路径长度。由于左子树中的节点成等概率分布，所以这棵树的左子树的内部路径长度为：

$$\frac{1}{N}\sum_{i=0}^{N-1} D(i)$$

同理可以得到这棵树的右子树的内部路径长度为：

$$\frac{1}{N}\sum_{i=0}^{N-1} D(N-i-1)$$

所以有：

$$D(N) = \frac{2}{N}\sum_{i=0}^{N-1} D(i)$$

如果一个节点的深度为d，那么为了搜索该节点，相应的操作过程中被访问的节点数目就应当是$d + 1$。例如，具有一个节点的二叉搜索树的根节点深度为 0，而相关操作过程中被访问的节点数目为 1。所以当：

$$D(N) = \frac{2}{N}\sum_{i=0}^{N-1} D(i)$$

二叉搜索树中所有节点的搜索操作时间消耗总和应当为：

$$D(N) = \frac{2}{N}\left[\sum_{i=0}^{N-1} D(i)\right] + N$$

由上式可得：

$$D(N) = 2\left[\frac{D(0) + D(1) + D(2) + \cdots + D(N-1)}{N}\right] + N$$

于是有：

$$ND(N) = 2[D(0) + D(1) + D(2) + \cdots + D(N-2) + D(N-1)] + N^2$$

$$(N-1)D(N-1)=2[D(0)+D(1)+D(2)+\cdots+D(N-2)]+(N-1)^2$$

将上述两式彼此相减得：

$$ND(N)-(N-1)D(N-1)=2D(N-1)+2N-1$$

下面整理上式并将末尾的–1 忽略不计得：

$$ND(N)=(N+1)D(N-1)+2N$$

整理得：

$$\frac{D(N)}{N+1}=\frac{D(N-1)}{N}+\frac{2}{N+1}$$

由上式亦可导出：

$$\frac{D(N-1)}{N}=\frac{D(N-2)}{N-1}+\frac{2}{N}$$

$$\frac{D(N-2)}{N-1}=\frac{D(N-3)}{N-2}+\frac{2}{N-1}$$

$$\cdots\ \cdots$$

$$\frac{D(2)}{3}=\frac{D(1)}{2}+\frac{2}{3}$$

将上面等式的左右两边分别相加得：

$$\begin{aligned}\frac{D(N)}{N+1}&=\frac{D(1)}{2}+2\left(\frac{1}{3}+\frac{1}{4}+\cdots+\frac{1}{N}+\frac{1}{N+1}\right)\\&=2\left(1+\frac{1}{2}+\frac{1}{3}+\cdots+\frac{1}{N}+\frac{1}{N+1}\right)-5=O(\log N)\end{aligned}$$

即：

$$D(N)=O(N\log N)$$

于是我们得到，在平均情况下，二叉搜索树中所有节点搜索成功的时间代价总和约为$O(N\log N)$。又因为二叉搜索树中有N个节点，所以平均每个节点的搜索操作时间复杂度应当接近于$O(\log N)$，即保持为一个对数级的复杂度。

当向一棵二叉搜索树中插入一个节点的时候，事实上就是在树中搜索该节点的过程，但是这个搜索必然是一个失败的搜索。无论是一个失败的搜索还是一个插入操作，最终要找到的就是一个空的 LeftChild 指针或者空的 RightChild 指针。一个具有N个节点的二叉搜索树所具有的空指针数为$N+1$, 该结论可在第 7 章中所列之定理的基础上推出，这里不再给出具体证明。可以从这些空指针的角度来考虑一个插入操作（或者一个失败的搜索操作）的时间复杂度。这里定义一棵二叉搜索树中的所有$N+1$个空指针的深度总和为该树的外部路径长度（EPL，External Path Length）。而对于一个内部路径长度为IPLTree的二叉树来说, 它的外部路径长度 EPL 应该为IPLTree $+2N$, 其中N是树中的节点数。由此很容易推得二叉搜索树中节点插入操作的时间复杂度仍然是一个

对数级的复杂度。但是就此推断二叉搜索树中各主要操作的复杂度平均为$O(\log N)$还是为之过早。因为二叉搜索树中的节点删除操作较比查找和插入都复杂得多，因此它的复杂度必然高于$O(\log N)$。

综上所述，不难推出二叉搜索树中各节点的操作复杂度应当接近于但高于$O(\log N)$，这就回归到了前面我们所给出的基于实验的结论，即二叉搜索树操作的平均效率要比其最好情况下的效率大约差 38%。

9.2 自平衡的二叉搜索树——AVL 树

9.1 节介绍了有关二叉搜索树的一些内容。二叉搜索树是基于折半查找思想设计的一种数据结构。通过分析，可知二叉搜索树的确可以在很大程度上提高搜索的效率。然而，尽管当二叉搜索树处于平衡状态的时候，其操作时间复杂度为$O(\log N)$，但当二叉搜索树是单支树时，其搜索效率则为$O(N)$。可见，二叉搜索树的平衡性是影响其操作效率的关键。由此出发，学者们设计了第一个平衡二叉搜索树，即 AVL 树。AVL 树作为第一个平衡的二叉搜索树，其影响非常深远，后来的很多平衡结构都借鉴了 AVL 树的设计思想。AVL 树最初是由前苏联数学家格奥尔吉·阿杰尔松-韦利斯基（Georgy Adelson-Velsky）与叶夫吉尼·兰迪斯（Evgenii Landis）于 1962 年共同提出的，因此得名 AVL 树。

9.2.1 AVL 树的概念

AVL 树就是平衡二叉搜索树，其准确的定义如下：一棵 AVL 树或者是空树，或者是具有下列性质的二叉搜索树——它的左子树和右子树都是 AVL 树，且左子树和右子树的高度之差的绝对值不超过 1。如图 9-7 所示即为一棵 AVL 树，树中略去了各节点的关键码。

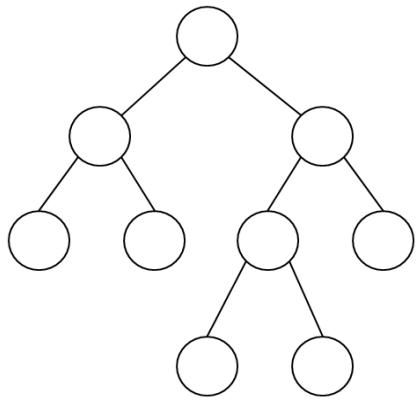


图 9-7 一棵 AVL 树

请读者注意，AVL 的平衡性是一种相对的平衡，而非一种绝对的平衡。与绝对平衡相比，这种所谓的相对平衡所满足的是一个较弱的平衡条件，即它不要求左子树和右子

树的高度绝对相等，而仅仅是左子树和右子树的高度之差的绝对值不超过 1 即可。之所以将平衡条件降低是因为绝对的平衡很难实现。

如果给 AVL 树中每个节点都附加一个数字，该数字指示该节点右子树的高度减去左子树的高度所得的高度差，那么这个数字即为该节点的平衡因子。根据 AVL 树的定义，任一节点的平衡因子只能取–1、0 或 1。如果一个节点的平衡因子的绝对值大于 1，则这棵二叉搜索树就失去了平衡，即不再是 AVL 树。如果一棵二叉搜索树是相对平衡的，它就是一棵 AVL 树。对于有n个节点的 AVL 树，其高度可保持在$\lfloor \log_2 n \rfloor$左右，平均搜索复杂度也可保持为$O(\log n)$。

9.2.2　AVL 树的旋转

算法

AVL 树的旋转算法。

作为一种平衡二叉搜索树，AVL 树的基本操作（如 find、insert 和 remove）与普通的二叉搜索树无异。但是其中某些操作有可能会破坏 AVL 树的平衡性，实时地保持这种平衡性非常重要。通常可以通过调整树的结构，使之保持平衡，这种用以进行平衡化处理的操作被称为旋转。

如果在一棵平衡的二叉搜索树中插入一个新节点，就会造成其失衡，这种失衡通常可归结为 4 种基本情况。

- 向某节点的左子树中插入一个左孩子。
- 向某节点的右子树中插入一个右孩子。
- 向某节点的左子树中插入一个右孩子。
- 向某节点的右子树中插入一个左孩子。

这 4 种情况所对应的结果分别如图 9-8 中的(a)、(b)、(c)、(d)4 幅子图所示。

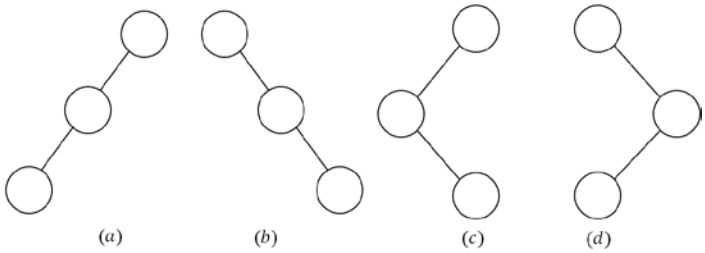


图 9-8　AVL 树失衡的 4 种基本情况

与上述 4 种失衡情况相对应的处理方式被分为两类，即单旋和双旋。其中，单旋又分为左旋和右旋，而双旋又分为先左后右和先右后左两种。那么在什么时候使用哪种方法处理失衡问题呢？通常每次向二叉搜索树中插入一个新节点时，AVL 树中相关节点的平衡状态就会发生改变。因此，在插入一个新节点后，就需要从插入位置沿通向根的路径回溯，检查各节点的平衡因子。如果在某一节点发现高度不平衡，则停止回溯。然后从发生不平衡的节点起，沿刚才回溯的路径取直接下两层的节点。这时大体的情

况分为两种，如果这 3 个节点处于一条直线上，如图 9-8 中的(a)、(b)两幅子图所示，则采用单旋进行平衡化。如果这 3 个节点不处于一条直线上，如图 9-8 中的(c)、(d)两幅子图所示，则采用双旋进行平衡化。

1．单旋

首先介绍单旋中的右旋方式。如图 9-9 所示，如果在左子树 *D* 上插入新节点使其高度增 1，而导致节点 *A* 的平衡因子增到–2，造成了不平衡。这时，为了使树恢复平衡，从 *A* 沿插入路径连续取 3 个节点 *A*、*B* 和 *D*，它们处于一条直线上，情况如图 9-8(a)所示，这时就需要做右单旋转。

右单旋转的方法是以 3 个成直线排列的节点中的中间节点为轴，进行顺时针旋转。该中间节点的原父节点将变成该节点的右子节点，该中间节点的右子树则变成其原父节点的左子树。仍以如图 9-9 所示的失衡 AVL 树为例，它的右单旋转过程如图 9-10 所示。可见，具体操作是以节点 *B* 为旋转轴，将节点 *A* 顺时针旋转，使其成为 *B* 的右子节点，而 *B* 的原右子树则变成了节点 *A* 的左子树。

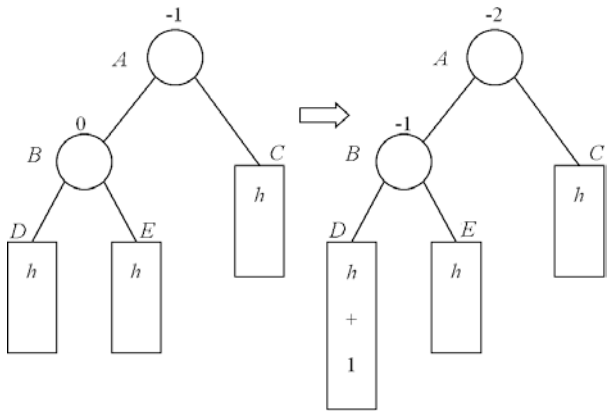


图 9-9　插入节点后 AVL 树失衡（情况 1）

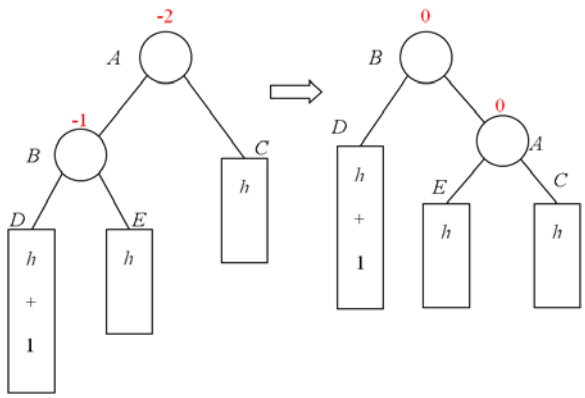


图 9-10　右单旋转

下面介绍单旋中的左旋方式。如图 9-11 所示，如果在右子树 *E* 上插入新节点使其高度

增 1，而导致节点 *A* 的平衡因子增加到 2，造成了不平衡。这时，为了使树恢复平衡，从 *A* 沿插入路径连续取 3 个节点 *A*、*C* 和 *E*，它们处于一条直线上，如图 9-8(*b*)所示，这时就需要进行左单旋转。

左单旋转的方法是以 3 个成直线排列的节点中的中间节点为轴，进行逆时针旋转。该中间节点的原父节点将变成该节点的左子节点，该中间节点的左子树则变成其原父节点的右子树。以如图 9-11 所示的失衡 AVL 树为例，它的左单旋转过程如图 9-12 所示。可见，具体操作是以节点 *C* 为旋转轴，将节点 *A* 逆时针旋转，使其成为 *C* 的左子节点，而 C 的原左子树则变成了节点 *A* 的右子树。

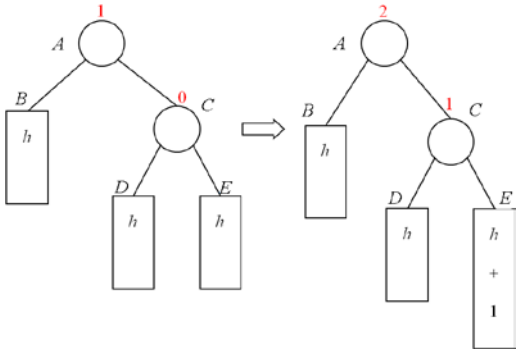


图 9-11　插入节点后 AVL 树失衡（情况 2）

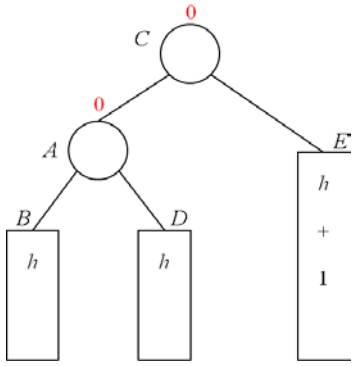


图 9-12　左单旋转

2．双旋

先左后右双旋转的处理方式用于解决形如图 9-8(*c*)中的情况。例如有如图 9-13(*a*)所示的一棵 AVL 树，在子树 *F* 或 *G* 中插入一个新节点，该子树的高度增 1。节点 *A* 的平衡因子变为–2，发生了不平衡，如图 9-13(*b*)所示。这时，为了使树恢复平衡，从节点 *A* 起沿插入路径选取 3 个节点 *A*、*B* 和 *E*，它们位于一条形如“ < ”的折线上，情况如图 9-8(*c*)所示，因此这时需要进行先左后右的双旋转以恢复 AVL 树的平衡性。

先左后右双旋转的处理方法是以 3 个成折线排列的节点中的末节点为轴，进行逆时针旋转。使末节点代替中间节点的位置，也就是让末节点成为原中间节点的父节点，而末节点的左子树则变成原中间节点的右子树。这时，3 个节点将成一直线排列，原来

的末节点变成了 3 条直线中的中间节点，而原来的中间节点则变成了 3 条直线中的末节点。这时再以新的中间节点为旋转轴做右单旋转，即可完成平衡化操作。

以如图 9-13(*b*)所示的失衡 AVL 树为例，它的先左后右双旋转过程如图 9-14(*a*)和 9-14(*b*)所示。可见，具体操作是以节点 *E* 为旋转轴，将节点 *B* 逆时针旋转，以节点 E 代替原来节点 *B* 的位置，也就是让节点 *E* 成为节点 *B* 的父节点，而节点 *E* 的左子树则变成节点 *B* 的右子树。这时，再以节点 *E* 为旋转轴，将节点 *A* 顺时针旋转，即做右单旋转，使之平衡化。

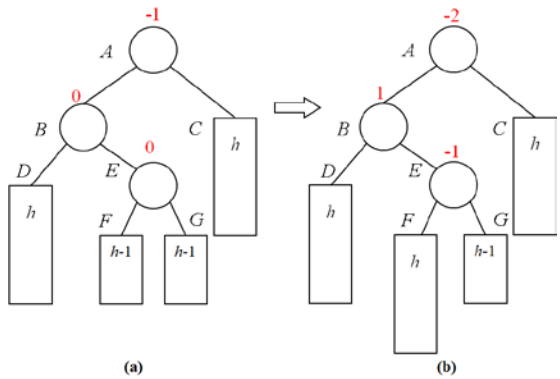


图 9-13　因插入操作而导致失衡（情况 1）

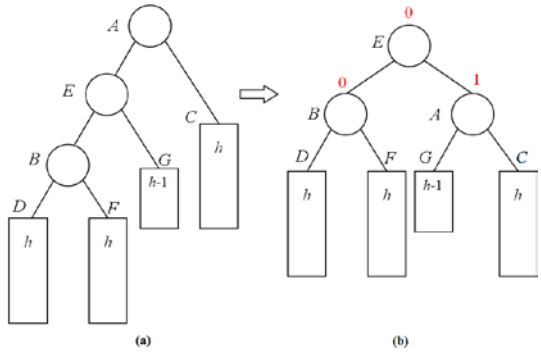


图 9-14　先左后右双旋转

先右后左双旋转是先左后右双旋转的镜像。先右后左双旋转的处理方式用于解决形如图 9-8(*d*)中的情况。例如，有如图 9-15(*a*)所示的一棵 AVL 树，在子树 *F* 或 *G* 中插入一个新节点，该子树的高度增 1。节点 *A* 的平衡因子变为 2，发生了不平衡，如图 9-15(*b*)所示。为了使树恢复平衡，从节点 *A* 起沿插入路径选取 3 个节点 *A*、*C* 和 *D*，它们位于一条形如“ > ”的折线上，情况如图 9-8(*d*)所示，因此这时需要进行先右后左双旋转以恢复 AVL 树的平衡性。

先右后左双旋转的处理方法是以 3 个成折线排列的节点中的末节点为轴，进行顺时针旋转。使末节点代替中间节点的位置，也就是让末节点成为原中间节点的父节点，而末节点的右子树则变成原中间节点的左子树。这时，3 个节点将成一直线排列，原来的末节点变成了 3 条直线中的中间节点，而原来的中间节点则变成了 3 条直线中的末

节点。这时再以新的中间节点为旋转轴做左单旋转，即可完成平衡化操作。

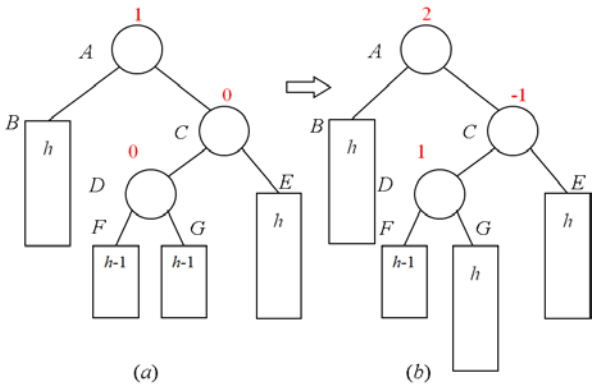


图 9-15　因插入操作而导致失衡（情况 2）

以如图 9-15(*b*)所示的失衡 AVL 树为例，它的先右后左双旋转过程如图 9-16(*a*)和图 9-16(*b*)所示。可见，具体操作是以节点 *D* 为旋转轴，将节点 *C* 顺时针旋转，以节点 *D* 代替原来节点 *C* 的位置，也就是让节点 *D* 成为节点 *C* 的父节点，而节点 *D* 的右子树则变成节点 *C* 的左子树。这时，再以节点 *D* 为旋转轴，将节点 *A* 反时针旋转，即做左单旋转，使之平衡化。

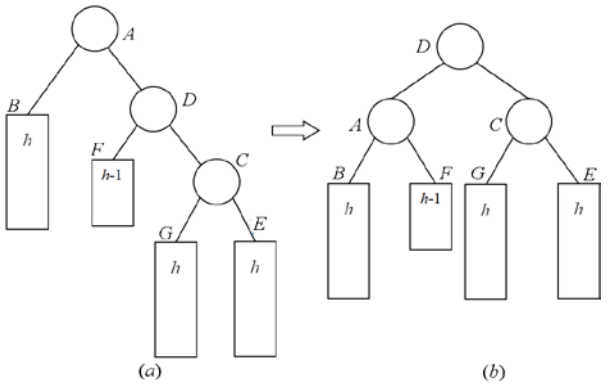


图 9-16　先右后左双旋转

AVL 树的一般操作与普通二叉搜索树没有太大的区别，只是为了维持其平衡性，AVL 树在进行操作的同时需要做相应的旋转操作。前面通过对添加一个节点后 AVL 树发生失衡的 4 种基本情况的讨论给出了相应的旋转处理方案。AVL 树的旋转操作是 AVL 树操作思想的核心和精华，因此掌握它非常重要。读者可以将上述情况和方法简单拓展一下，并考虑如何通过平衡化操作来在删除某节点时使树的平衡性不变。

9.2.3　AVL 树的实现

本节编程实现一个 AVL 树类。AVL 树类与二叉搜索树类非常相似，节点类 AvlNode 较普通二叉搜索树节点类多了一个成员变量 balance，它将用来表示平衡因子。AVL

树类中比较重要的地方在于旋转操作的实现函数。下面先给出节点类 AvlNode 的类定义，它位于头文件“ avlnode.h ”中。其中将 AVL 树类 AvlTree 声明为友元类，AvlTree 类和 AvlNode 类都是模板类。

```
#ifndef AVLNODE_H
#define AVLNODE_H

#include <iostream>

using namespace std;

template <class T> class AvlTree; //声明 AvlTree 类

template <class T>
class AvlNode{

    T data;                   //关键码
    AvlNode *leftChild;       //左孩子
    AvlNode *rightChild;      //右孩子
    int balance;              //平衡因子

public:

    //构造函数
    AvlNode():left(NULL),right(NULL),balance(0){};

    AvlNode(const T& e,AvlNode<T> *lt = NULL,AvlNode<T> *rt = NULL)
        :data(e),leftChild(lt),rightChild(rt),balance(0){};

    int getBalance() const
    {
        return balance;
    }
    AvlNode<T>* getLeftChild() const
    {
        return leftChild;
    }
    AvlNode<T>* getRightChild() const
    {
        return rightChild;
    }
    T getData() const
    {
        return data;
    }

    friend class AvlTree<T>;
};
#endif
```

下面给出 AVL 树类 AvlTree 的类定义，它位于头文件“ avltree.h ”中。

```
template <class T>
class AvlTree{

    AvlNode<T> *root;

    bool Insert(AvlNode<T> *& rt,T x, bool &taller);
```

```
    bool Remove(AvlNode<T> *& rt,T x, bool &shorter);

    void RotateLeft(AvlNode<T> * &node);    //左旋函数
    void RotateRight(AvlNode<T> * &node);   //右旋函数

    void RightBalanceAfterInsert(AvlNode<T> * &sRoot, bool &taller);
    void LeftBalanceAfterInsert(AvlNode<T> * &sRoot, bool &taller);
    void RightBalanceAfterDelete(AvlNode<T> * &sRoot, bool &shorter);
    void LeftBalanceAfterDelete(AvlNode<T> * &sRoot, bool &shorter);

public:

    AvlTree():root(NULL){}
    AvlNode<T>* getRoot() const
    {
        return root;
    }

    bool Insert(T x)
    {
        bool taller = false;
        return Insert(root, x, taller );
    }

    bool Remove(T x)
    {
        bool shorter = false;
        return Remove(root, x, shorter);
    }

    //输出树形结构
    void DisplayTree(AvlNode<T> *t,int layer) const;
};
```

AvlTree 树类中比较重要的成员函数是左旋函数、右旋函数、节点插入函数和节点删除函数。限于篇幅这里不能将它们全部列出，下面仅给出左旋函数和右旋函数的实现代码。读者可以从本书的在线支持网站上下载得到该类的完整实现代码。

```
template <typename T>
void AvlTree<T>::RotateLeft(AvlNode<T> * & node)
{
    if( (node == NULL) || (node->rightChild == NULL) ) return;

    AvlNode<T> * tmpNode = new AvlNode<T>(node->data);
    if(tmpNode == NULL ) return;

    tmpNode->leftChild = node->leftChild;
    node->leftChild = tmpNode;
    tmpNode->rightChild = node->rightChild->leftChild;

    AvlNode<T> *toDelete = node->rightChild;
    node->data = toDelete->data;
    node->rightChild = toDelete->rightChild;

    delete toDelete;
}

template <typename T>
void AvlTree<T>::RotateRight(AvlNode<T> * & node)
```

```
{
    if( (node == NULL) || (node->leftChild == NULL) ) return;

    AvlNode<T> *tmpNode = new AvlNode<T>(node->data);
    if(tmpNode == NULL ) return;

    tmpNode->rightChild = node->rightChild;
    node->rightChild = tmpNode;
    tmpNode->leftChild = node->leftChild->rightChild;

    AvlNode<T> *toDelete = node->leftChild;
    node->data = toDelete->data;
    node->leftChild = toDelete->leftChild;

    delete toDelete;
}
```

下面给出用于测试上述类的一个示例程序。该程序首先建立一个 AVL 树，可以通过输入一定的关键码来选择要删除的节点。程序的输出结果是每次经过调整后的 AVL 树，它没有采用通常的中序、前序或者后序遍历方法而是采用了一种更加直观形象的方式来将二叉树打印到屏幕上。

```
#include <conio.h>
#include <iostream>
#include "avltree.h"

using namespace std;

void main()
{
   AvlTree<int> tree;

    tree.Insert(21);
    tree.Insert(12);
    tree.Insert(35);
    tree.Insert(0);
    tree.Insert(15);
    tree.Insert(36);
    tree.Insert(14);
    tree.Insert(7);
    tree.Insert(28);
    tree.Insert(9);

   tree.DisplayTree(tree.getRoot(), 0);

    while(true)
    {
        int number;
        cout<<"请输入要删除的节点:"<<endl;
        cin>>number;
        tree.Remove(number);
        cout<<"删除后的树为:"<<endl;
        tree.DisplayTree(tree.getRoot(), 0);

        cout<<"按 Esc 键退出，或按任意键继续!"<<endl;
        int order=getch();
        if (order ==27)
        {
            break;
```

```
        }
    }
}
```

完成编码后编译并运行程序即可。

9.3　树中亦有“红与黑”

红黑树是一种自平衡的二叉搜索树，也就是一种特殊的二叉树，它同样是计算机科学中一种重要的数据结构。红黑树最早是由德国计算机科学家鲁道夫·拜尔（Rudolf Bayer）于 1972 年发明的。当时，拜尔把这种结构称为“对称二叉 B 树”。红黑树这个名字则源于后来里奥·古贝斯（Leo J. Guibas）和罗伯特·塞奇威克（Robert Sedgewick）于 1978 年完成的一篇论文。红黑树的结构和操作都比较复杂，但优点在于其操作有着良好的最坏情况运行时间。大量实践证明红黑树是高效的，本节将对红黑树的相关内容进行介绍。

9.3.1　红黑树的概念

红黑树是一种特殊的二叉搜索树。当一棵二叉搜索树满足下面的性质时，它就是一棵红黑树。

（1）每个节点或是红的，或是黑的。

（2）每个叶节点（NIL）都是空节点，并且被着为黑色。

（3）如果一个节点是红的，则它的两个子女都是黑的。

（4）从某一节点到达其子孙叶节点的每一条简单路径上包含相同个数的黑节点。

（5）根节点永远是黑色的。

关于上述性质不同文献上的描述存在差异，这里需要进行简单的说明。在此之前，先来看看红黑树节点的组成结构。为了组建红黑树的结构，原来二叉树中每个节点都包含有 5 个域：color、key、left、right 和 p，如图 9-17 所示。其中，color 域指示该节点的颜色，key 域中存储该节点的值，left 域指示该节点的左子节点，right 域指示该节点的右子节点，p 域指示该节点的父节点。

图 9-17　红黑树的节点域

如果红黑树中某节点的一个子节点或父节点不存在，那么该节点相应的指针域包含值 NIL，并且将把这些 NIL 看作是指向二叉搜索树的外节点的指针，而把带关键字的普通节点看作是树的内节点。

参考文献[4]中给出的红黑性质是不包括第（2）条的。这看上去与本书所给出的红黑

性质有很大出入。但是二者并不矛盾，而只是两种不同的表述方式。注意本书所给出的红黑性质第（2）条表示每个叶子节点均默认为黑色并置为空，这表示实际这些节点并不存在，它们仅仅是其父节点的一个空 leftChild 或空 rightChild 指针。而参考文献[4]中所给出的红黑树均是忽略了这些空节点的。之所以将这些空节点都涂为黑色是为了使性质（4）能在更强的条件下成立。也就是说在实际考察红黑树时是可以忽略掉这些 NIL 节点的。特别在考察红黑树的插入或删除操作时可以这样处理，这将有助于理解。

易见红黑树是二叉搜索树，但其每个节点上都需要增加一位存储位用来表示该节点的颜色，且具体颜色只能是红色（RED）或者黑色（BLACK）的。

此外，由于红黑树是二叉搜索树，所以其中每个节点的比较值都必须大于或等于该节点的左子树中的所有节点，并且小于或等于它右子树中的所有节点。如此才能保证红黑树运作时可以快速地在树中搜索到给定值。

如图 9-18 所示为一棵红黑树。其中黑节点以暗色标出，红节点以白色标出。一棵红黑树中的每个节点或者是黑的，或者是红的。其中根节点和所有叶子节点都是黑色的。一个红节点的所有子女都是黑的。每一条从一个节点到其后代叶节点的简单路径都包含了相同数目的黑节点。在本例中从根节点到任何叶子节点的简单路径中都有且只有 3 个黑节点。

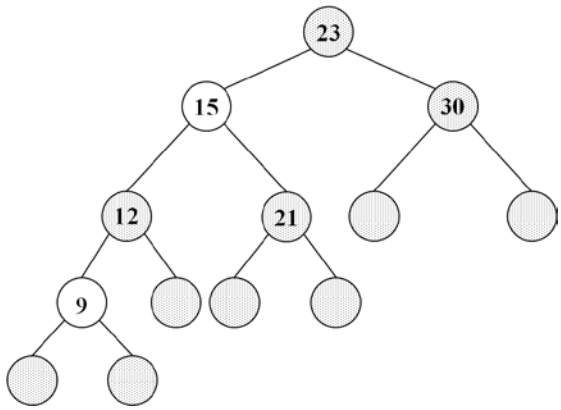


图 9-18　红黑树

之所以说红黑树是平衡的，是因为红黑规则通过对任意一条从根到叶子的路径上各节点的着色方式加以限制，从而确保红黑树中不存在一条路径会比任何其他路径长到两倍，因而红黑树基本上是平衡的。

红黑树中的一个重要概念是黑高度。将红黑树中从某个节点x出发（不包括该节点）到达一个叶节点的任意一条路径上的黑节点个数称为该节点的黑高度，用$\mathrm{bh}(x)$表示。根据上面的性质（4）可知从红黑树中某节点出发的所有下降路径都有相同的黑节点个数。据此可以定义红黑树中根节点的黑高度为该树的黑高度。通常红黑树中的每个非叶子节点都被标以其黑高度，而叶子（NIL）节点的黑高度为 0。

红黑树是一种很有效的平衡搜索树。它和 AVL 树一样都对插入时间、删除时间和查找

时间提供了最好可能的最坏情况担保。但同时红黑树的统计性能要好于 AVL 树，一方面红黑树在时间敏感的应用（如即时应用）中能很好地发挥作用。此外，它也在提供最坏情况担保的其他数据结构中被用来作为建造板块。在 C++ STL 中的很多部分，例如 set、multiset、map 以及 multimap 都应用了红黑树的变体提供支持。此外，红黑树在函数式编程中也是最常用的持久数据结构之一，它被用来构造关联数组和集合，并保持关联数组和集合在发生突变之后仍为以前的版本。

红黑树之所以有效，因为它能保证在最坏情况下动态集合操作（查找、插入和删除）的时间为$O(\log n)$，其中n是树中元素的数目。为了说明这一点，先来证明一个结论：一棵含有n个内节点的红黑树的高度至多为$2\log(n+1)$。

证明：为了证明上述结论，先来证明以某一节点x为根的子树中至少包含$2^{\mathrm{bh}(x)}-1$个内节点。这里使用数学归纳来证明。

归纳基础：当x的高度为 0，则x必然为一个叶节点（NIL），这时以x为根的子树包含$2^{\mathrm{bh}(x)}-1=2^0-1=0$个内节点，所以命题成立。

归纳假设：对于所有的$k>0$，以k为根的子树中至少包含$2^{\mathrm{bh}(x)}-1$个内节点。

归纳步骤：一个其高度为正值的节点x，它是个内节点且有两个子女。其中每个子女的颜色可以是黑也可以是红。如果其子节点是黑的，则该子节点的黑高度是$\mathrm{bh}(x)-1$。如果其子节点是红的，则该子节点的黑高度是$\mathrm{bh}(x)$。又因为x子女的高度都小于x的高度，根据归纳假设可得其每个子女至少包含$2^{[\mathrm{bh}(x)-1]}-1$个内节点。这样以x为根的子树至少包含的内节点数为：

$$2^{[\mathrm{bh}(x)-1]}-1+2^{[\mathrm{bh}(x)-1]}-1+1=2^{\mathrm{bh}(x)}-1$$

即该结论成立。

下面继续证明待证之结论。性质（3）告诉我们如果一个节点是红的，则它的两个子女都是黑的。这说明在红黑树中，从每个叶子到根的所有路径上不能有两个连续的红色节点。因此很容易证明，红黑树中能够建立的最长路径将是一条红黑交替的路径，而红黑树中能够建立的最短路径则是一条只有黑色节点的路径。即对于给定的黑色高度为$\mathrm{bh}(x)$的红黑树，从根节点到叶节点的简单路径的最短长度为$\mathrm{bh}(x)$，最大长度为$2[\mathrm{bh}(x)]$。设树的高度为h，那么可得$\mathrm{bh}(x)\leqslant h\leqslant 2[\mathrm{bh}(x)]$，即根的黑高度至少为$h/2$。若树中有$n$个内节点，又因为以某一节点$x$为根的子树中至少包含$2^{\mathrm{bh}(x)}-1$个内节点，所以有$n\geqslant 2^{h/2}-1$。把 1 移到不等号左边，再对不等式两边取对数，得$\log(n+1)\geqslant h/2$，即$h\leqslant 2\log(n+1)$。综上所述，结论得证。

对于一棵高度为h的二叉搜索树来说，在其上完成搜索、取最小值、取最大值、取某节点前驱节点以及取某节点后继节点等操作的时间复杂度都是$O(h)$，而包含n个内节点的红黑树满足$h\leqslant 2\log(n+1)$。所以在红黑树上进行上述操作的时间复杂度都将是$O(\log n)$。除此之外，由于红黑树在进行删除和插入操作后能够通过一定的调整继续保持红黑性质，因此在红黑树上进行这些操作的时间复杂度仍为$O(\log n)$。

9.3.2 红黑树的操作

本节介绍与红黑树有关的一些操作。这些操作的共同特点在于必须保证操作不会破坏原树的红黑性质，为此所有这些操作都基于一些调整来保持红黑树的性质。

算法

红黑树的旋转算法。

1. 旋转

当在红黑树上进行插入或删除操作时，树的红黑性质将被破坏。为了保持这些性质，就要改变树中某些节点的颜色以及指针的结构。指针结构的修改是通过旋转来完成的。本书在介绍 AVL 树时曾经介绍过旋转操作，在简单复习一下之后我们将提供一个具体的实例来帮助读者加深理解。

旋转是一种能保持树中节点的中缀次序的局部操作。图 9-19 演示了两种旋转：左旋转和右旋转。其中，右旋转操作RIGHT_ROTATE(T, x)将左边两个节点的结构通过改动若干指针而变为右边的结构。通过其逆操作，以及左旋转操作LEFT_ROTATE(T, y)又可将右边的结构改为左边的结构。这两个节点可以出现于树中的任何位置。字母α、β和γ表示任意子树。旋转保持了节点之间的中缀次序：α子树中的根节点前于节点x，又前于β子树中的根节点，β子树中的根节点又前于节点y，它又前于γ子树中的根节点。

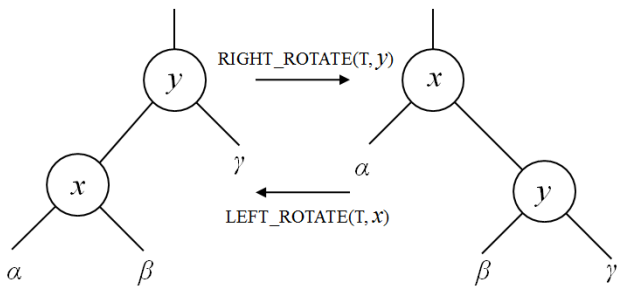


图 9-19 旋转

当将某个节点x左旋时，假设其右孩子y是非 NIL 的，且旋转以x到y之间的链为轴来进行。左旋的结果使得y成为子树新根，而x成为y的左孩子，y的左孩子也将成为x的右孩子。

图 9-20 展示出了左旋操作过程，注意树中略去了 NIL 叶子。经过左旋操作，原节点 30 的右子节点 35 成为子树的新根。而节点 30 则变成了节点 35 的左子节点。节点 35 的原左孩子也成为了节点 30 的右孩子。

可见，经过左旋转，二叉搜索树中节点的比较值关系仍然正确，即其中每个节点的比较值都大于或等于该节点的左子树中的所有节点。另外，读者可以试着写出图 9-20 中操作前后两棵树的中序遍历结果，二叉树的中序遍历在第 8 章中已经讲过，这里不再赘述。观察二者的中序遍历结果，不难发现，输入树及修改后的树的中序遍历产生结果是相同的。根据等价关系的对称性，易知右旋操作的情况也一样。注意，这两种旋转操作时间复杂度都为$O(1)$。在旋转操作中被改变的仅是指针，而节点中其他域则保持不变。

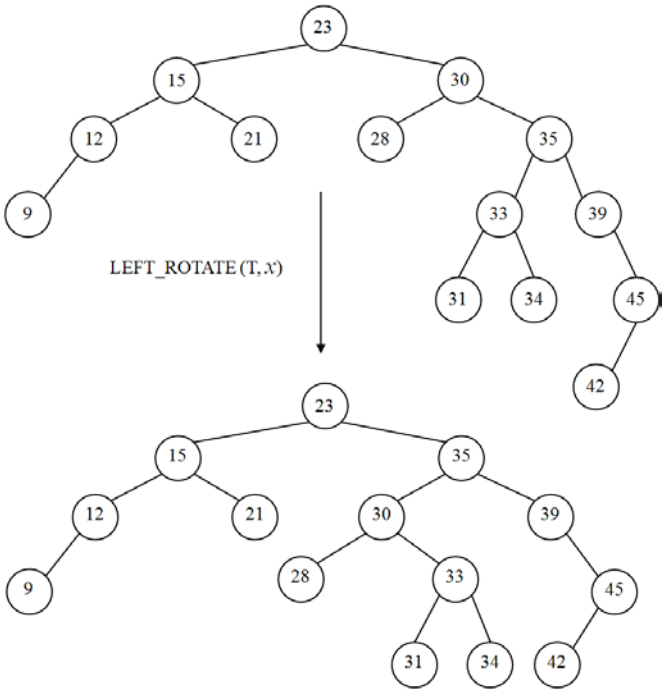


图 9-20　左旋操作

2．插入

算法

红黑树的节点插入算法。

向一棵红黑树中插入一个节点的过程是这样的：先使用普通二叉搜索树的节点插入算法将新节点插入树中，然后将该节点涂为红色。为了不破坏红黑性质，需要对有关节点进行重新着色并旋转。因为向二叉搜索树中插入一个元素其实就是向其中添加一个叶子节点（注意这里忽略红黑树的 NIL 节点），所以在向红黑树中插入一个元素时就必须将其涂为红色。如果不是这样，势必会违反红黑性质（4），即如果新插入的叶子节点被涂为黑色，那么从某一节点到达其子孙叶节点的每一条简单路径上所包含的黑节点数目必然不相等。

但是接着问题又来了，如果新插入节点的父节点为黑色，那么红黑性质没有被破坏，操作就此完成。但是如果新插入节点的父节点为红色，那么就违反了红黑性质（3）。这时情况可能变得复杂多了，但可以将其归结为如下 3 种基本情况并加以处理。

情况 1：如图 9-21 所示，X为新插入的节点，将其涂为红色。P为X的父节点，亦为红色。G是X的祖父节点，S是X的叔父节点，它们都为黑色。这时X、P和G成一直线排列。此种情况的处理方式是采用一个单旋转操作并进行适当的颜色调整即可。如图 9-21 所示的情况，首先进行一个右单旋转，然后修改G的颜色，结果表示该树仍满足红黑性质。

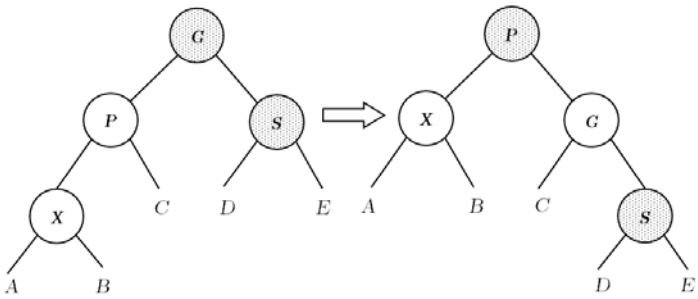


图 9-21　红黑树节点插入（情况 1）

情况 2：如图 9-22 所示，X为新插入的节点，将其涂为红色。P为X的父节点，亦为红色。G是X的祖父节点，S是X的叔父节点，它们都为黑色。这时X、P和G成一折线排列。此种情况的处理方式是采用一个双旋转操作并进行适当的颜色调整即可。如图 9-22 所示的情况，首先进行一个先左后右双旋转，旋转过程中修改X和G的颜色，结果表示该树仍满足红黑性质。

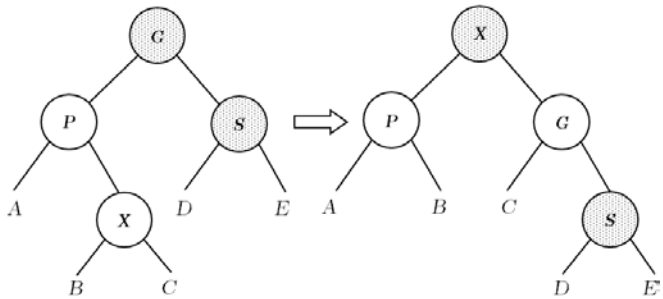


图 9-22　红黑树节点插入（情况 2）

情况 3：情况 1 和情况 2 都相对比较简单，但是如果新插入节点X的叔父节点S为红色的话，情况就变得更复杂了。情况 1 和情况 2 所面对的问题在经过简单旋转和颜色调整后即可被解决。但是如果新插入节点的叔父节点为红色的话，那么就有可能无论如何旋转总是会出现“红—红节点”的情况。这时所采用的解决方法就是逐层上升式的调整和着色。

如图 9-23 所示的情况，似乎一个右单旋转就解决了问题。但是如果节点X的祖父节点的父节点仍为红色呢？这时我们将不得不向根节点的方向逐层上升同时调整并着色。直到树中不再出现“红—红节点”的情况。

图 9-24 展示出了红黑树节点插入过程的一个例子，图 9-24(a)中插入了一个节点 16。因为 16 及其父节点 17 都是红的，故违反了性质（3）。又因为 16 的叔父节点 22 是红的，则属于情况 3。各节点被重新着色，指针沿树上升，所得的树如图 9-24(b)所示。如此，节点 21 及其父节点又都为红色，但节点 21 的叔父节点 30 是黑的。由于 21 是 15 的右孩子，故可应用情况 2 进行处理。在执行一次左旋转之后，所得结果树见图 9-24(c)。执行一次右旋后得图 9-24(d)中的树，它是一棵合法的红黑树，插入操作就此完成。

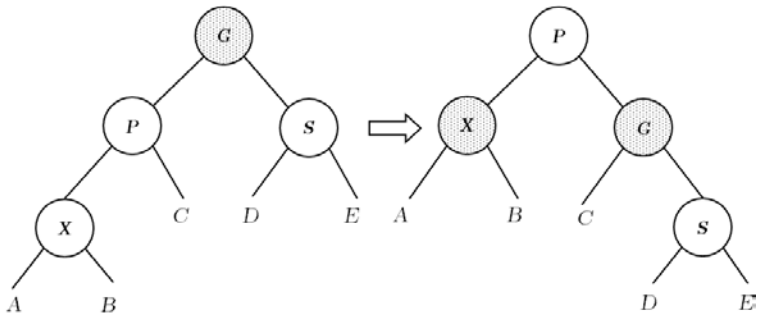


图 9-23　红黑树节点插入（情况 3）

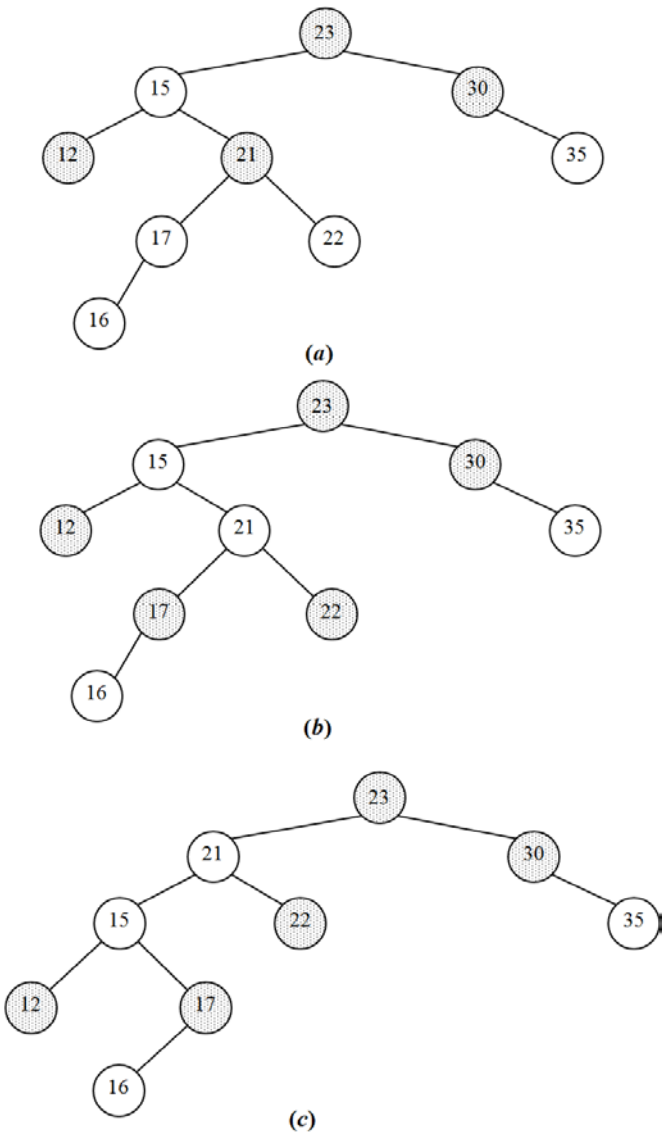
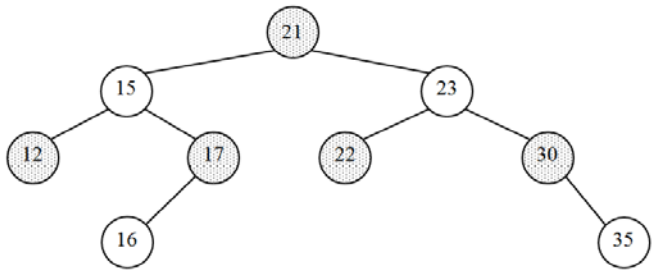


图 9-24　红黑树节点插入



(*d*)

图 9-24　红黑树节点插入（续）

算法

红黑树的节点删除算法。

3．删除

二叉搜索树的节点删除总是显得异常复杂。红黑树也不例外。先回想一下我们是如何从一棵普通二叉搜索树中删除一个节点的。为了从一棵普通二叉搜索树中删除一个节点，我们需要面对 3 种情况。

（1）待删除节点是没有子节点的，即它是一个叶子节点。在这种情况下，只需直接将其删除即可。

（2）待删除节点有一个子节点，这时可以直接将其删除。如果它是根节点，那么它的子节点变为根节点；相反，如果它不是根节点，那么将它的父节点中相应的子节点指针赋值为被删除节点的子节点的指针即可。

（3）如果待删除节点有两个子节点，则情况稍显复杂。首先找到该节点的后继节点（其右子树中关键码最小的那个节点），然后将这两个节点对调，之后将待删除节点删除。由于原后继节点不可能有左子节点，因此对调后的待删除节点相应也不会有左子节点，因此如果将它删除，那么删除该节点后的操作必然会落入情况（1）或情况（2）之一，继续做相应处理即可。

下面来讨论从红黑树中删除一个节点后可能遇到的情况，假设待删除节点为V，其父节点为P。注意从红黑树中删除一个节点的方法就是在节点删除（通过普通二叉搜索树节点删除方法）后，针对不同情况对红黑树进行调整以保持其红黑性质的过程。

若红黑树 T 是一棵空树，那么删除操作将执行空操作。若它只有根节点一个节点，那么该节点的删除方法只是简单地将树置为空即可。排除上述两种情况，易知节点V的子节点中至少有一个为叶节点。如果V的子节点均是叶子节点，那么将其删除的结果是它的位置最终将被其一个叶节点取代。否则，如果V有一个非叶子节点，那么将其删除的结果是它的位置将被这个非叶子节点取代。这里用X来表示红黑树删除操作后在树中取代了节点V位置的那个节点。如果X是叶节点，那么就可以确定它是黑色的（性质（2））。

如果节点V是红色的，那么删除操作就完成了。因为这种删除不会破坏红黑树的任何性质。如果节点V是黑色的，那么删除了V之后，从根节点到V的所有子孙叶节点的路径将会比树中其他的从根节点到叶节点的路径拥有更少的黑色节点，即破坏了红黑性

质（4）。此外，如果P与X都是红色的，那么性质（3）也会遭到破坏。这似乎显得情况异常复杂，但事实上，在解决性质（4）遭到破坏的过程中，性质（3）遭到破坏的问题是可以被同时解决的。所以我们所需面对的问题仅仅是如何处理性质（4）遭到破坏的问题。

为了便于理解，我们用一个记号来表示从根节点到这个带记号节点的所有子孙叶节点的路径上都缺少一个黑色节点。而我们的调正过程就是将该记号一直向树根方向移动，直到性质（4）恢复的过程。当然，这个标记只是我们假想出来的，它并不真实存在于树的结构中。

可能遇到的基本情况被分为 4 种。

情况 1：如果带标记的节点是红色的或者它是树的根节点，或者两者皆是，那么只要将它染为黑色就可以完成删除操作。显然，性质（4）就这样被恢复了，因为这个记号表示从根节点到该节点的所有子孙叶节点的路径都缺少一个黑色节点，如果将这个红色节点改变为黑色，那么就相当于在这些缺少一个黑色节点的路径上添加了一个黑色节点。同样因为一个红色节点消失了，那么红—红节点的情况也被克服了，即性质（3）得以恢复。

如果带记号的节点是根节点并且为黑色，那么直接将这个标记丢掉即可。因为在这种情况下，树中每条从根节点到叶节点的路径的黑色节点数量都比删除操作前少了一个，也就是说红黑性质（4）依旧得以保持。

情况 2：将节点V删除后，节点P和X就会连在一起，如果它们都是黑色节点，且X的兄弟节点以及两个侄子节点都是黑色的，那么就要将它的兄弟节点涂为红色之后再将标记向树根的方向移动一步。

如图 9-25 所示，X的兄弟节点为黑色，X的侄子节点C和D亦为黑色，节点P被涂成了灰色表示我们并不关心它的颜色。注意带有标记的黑色节点X必然有两个非叶子节点的侄子节点，即C和D必然存在。因为这个节点X的标记表示从根节点到该节点的所有子孙叶节点的路径中的黑色节点数量都比其他的根节点到叶节点路径所包含的黑色节点数量少 1。而X节点本身就是一个黑色节点，因此从它的兄弟节点到其子孙叶节点的路径上的黑色节点数量必然要大于 1，这就要求其兄弟节点的任何一个子节点都不能为叶节点。于是X节点必然会有两个非叶节点的侄子节点。

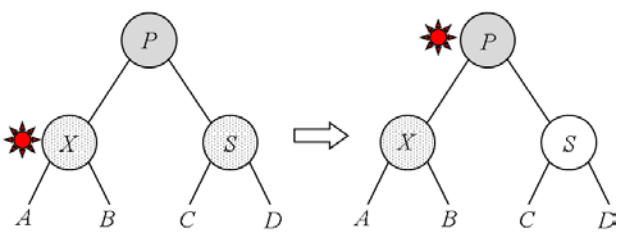


图 9-25　红黑树节点删除（情况 2）

将S节点染为红色之后，就相当于从所有到S节点的子孙叶节点的路径上去掉一个黑色

节点，因此现在这些路径上的黑色节点数量与到X节点的子孙叶节点的路径上的黑色节点数量相等。然后，需要将标记向上移动到P，这表明现在所有从根节点到P的子孙叶节点的路径上都缺少一个黑色节点。如果P不是根节点，那么问题就仍然没有解决，这时继续处理直到将标记推进到根节点为止。可见，每次处理实际都是将标记向根节点推进一个节点的过程。

此外，这种情况严格要求带标记之节点的两个侄子节点都是黑色的，因为如果它有一个侄子节点是红色的，那么将导致该操作的结果出现“红—红节点”的情况。

情况 3：情况 2 要求带标记节点X的兄弟节点是黑色的。而如果带标记之节点的兄弟节点是红色的，就是现在将要讨论的情况 3。这时，需要进行一次旋转操作并改变节点颜色。

图 9-26 演示了两种子情况的处理方式。如图 9-26(a)所示，当节点X的兄弟节点S为红色节点，且X是S的左兄弟节点时，就使用左旋处理方式，并修改节点P和节点S的颜色。如图 9-26(b)所示，当节点X的兄弟节点S为红色节点，且X是S的右兄弟节点时，就使用右旋处理方式，并同样修改节点P和节点S的颜色。

读者可能会发现一个问题，那就是这样的操作似乎并没有改变从根节点到任何叶节点路径上的黑色节点数量，也就是说好像操作没有对恢复性质 4 起到良性的作用。但事实上并非如此。经过该操作之后，节点X的兄弟节点必然是黑色的，这使得后续的操作归属于情况 2 或者情况 4 中的任意一个。读者应该已经注意到了操作后节点X的父节点是红色的，所以如果下一步操作属于情况 2，那么标记将会向上移动到红色节点P，这时只需将P染为黑色就完成操作。而对于即将介绍的情况 4，也总是可以通过一定的操作恢复树的红黑性质。因此，该操作的确对恢复性质 4 起到了推进作用。

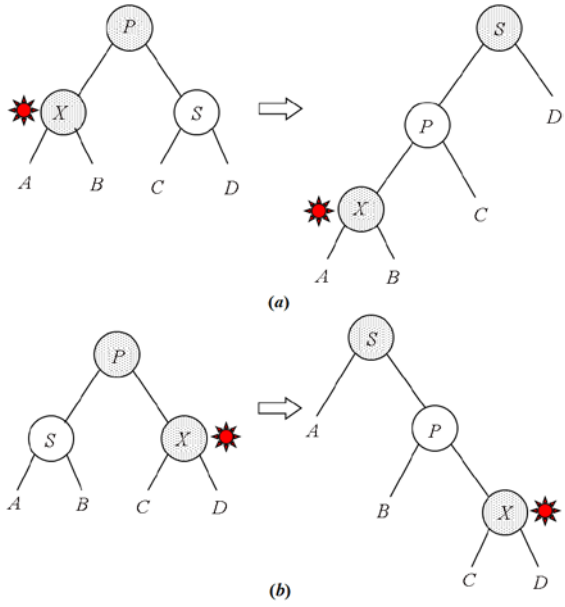


图 9-26　红黑树节点删除（情况 3）

情况 4：如果当节点*X*兄弟节点*S*为黑色，且*X*节点至少有一个侄子节点是红色的，那么它属于我们即将讨论的第 4 种情况。

为了便于描述，下面给出两个定义。

近侄子节点：如果二叉树中，某节点 Node 是其父节点的左子节点，那么 Node 节点兄弟节点的左子节点就是其近侄子节点。否则，如果节点 Node 是其父节点的右子节点，那么 Node 节点兄弟节点的右子节点为其近侄子节点。

远侄子节点：如果二叉树中，某节点 Node 是其父节点的左子节点，那么 Node 节点兄弟节点的右子节点就是其远侄子节点。否则，如果节点 Node 是其父节点的右子节点，那么 Node 节点兄弟节点的左子节点为其远侄子节点。

如果带标记节点*X*的远侄子节点是黑色的，那么在第 1 种子情况下它的近侄子节点一定是红色的；在第 2 种子情况下，其远侄子节点是红色的，这时它的近侄子节点可以是任何颜色。如图 9-27 所示，第 1 种子情况通过一次旋转和颜色调整即可转换为第 2 种子情况。而在第 2 种子情况下只要通过一次旋转和颜色调整就可以使红黑性质恢复。图 9-27 是根据带标记节点*X*是其父节点的左子节点还是右子节点而分别给出的两种对称的形式。

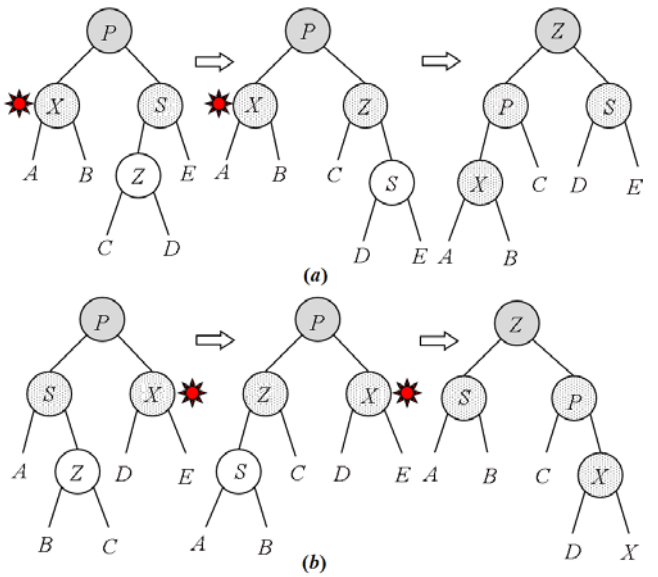


图 9-27　红黑树节点删除（情况 4）

可见操作之后，所有到带标记节点的子孙叶节点的路径上黑色节点数量都增加了 1，而其他的路径上的黑色节点数量保持不变。最终，红黑性质得以保持。

对上述 4 种基本情况下的红黑树进行具体调整，能够保证节点被删除后，二叉搜索树依旧满足红黑性质。因此，红黑树的节点删除过程就是首先调用普通二叉搜索树的节点删除方法，然后通过调整方法将红黑性质再次恢复的过程。

9.3.3 红黑树的实现

本节编程实现一个红黑树类。下面先给出红黑树类 RedBlackTree 的类定义，它是一个模板类，注意这里仅给出一些比较主要的成员函数的声明。

```
template<typename T>
class RedBlackTree
{
    class Node
    {
    public:
        T key;
        bool   color; //RED - 1, BLACK - 0
        Node*   parent;
        Node*   leftChild;
        Node*   rightChild;
        Node(T data, bool color, Node* p, Node* l, Node* r)
            :key(data),color(color),parent(p),
             leftChild(l),rightChild(r){}
   };

    Node* NIL;
    Node* root;

public:
    RedBlackTree(T data=0)
    {
        NIL = new Node(0, false, NULL, NULL, NULL);
        root = NIL;
    }

    Node* GetRoot(){return root;}
    Node* GetMin(Node* x);
    Node* FindElement(Node* x, T k);

    void  LeftRotate(Node* x);
    void  RightRotate(Node* y);
    void  RBInsert(T data);
    void  RBInsertFixup(Node* z);
    void  RBDelete(T data);
    void  RBDeleteFixUp(Node* x);
    //其他成员方法略
};
```

上述类中成员函数的实现代码可以从本书的在线支持资源中获得，这里不再赘述。

本书的在线支持资源中提供了红黑树类中比较重要的旋转操作函数和节点插入函数。至于节点删除函数的实现代码就留给有兴趣的读者自己完成。读者也可以到本书的在线支持资源中下载得到完整的红黑树类实现代码。

9.4 基于 Trie 树的单词检索

Trie 树又称为字典树或单词查找树，之所以称其为 Trie 树，是从 retrieve（检索）一词中取中间 4 个字符而形成的一个词。Trie 树是一种用于快速字符串检索的多叉树结构，

它的每一个节点代表了一个字符，而它的每一条边就代表了字符串的连接关系。Trie 树能够用于保存大量的字符串，并利用字符串的公共前缀来降低时空开销，从而达到提高程序效率的目的。

9.4.1　Trie 树的概念

Trie 树是一种非常有用的树形数据结构。它是一棵度$m \geqslant 2$的树，它的每一层分支不是靠整个关键码的值来确定的，而是由关键码的一个分量来确定的，即树中的每个节点仅仅包含组成关键码的符号而非关键码本身。

如图 9-28 所示为一棵 Trie 树，由于关键码是英文单词，故其节点只包含单独的字母。在这个结构中存储着单词表{blue, blur, car, cat, caw, coin}。在此 Trie 树的第i层，所有的关键码根据它们第i位字符，被划分到互不相交的 27 个子部分中。所以整个结构中的每一棵子 Trie 树都表示以其根节点中关键码开头的一个字符串。Trie 树的第 0 层(即根节点)为空。图 9-28 中的 Trie 树在第 1 层被分成了两组，即{blue, blur}和{car, cat, caw, coin}，而{car, cat, caw, coin}又被分为{car, cat, caw}和{coin}两组。显然，Trie 树将集合、子集以及元素之间的层次关系表示得非常清楚，这也为搜索带来了便利。与关键码是字符串的情况相同，如果 Trie 树中的关键码是数值，那么树中每个节点就应当只包含一个数位。

可以归纳 Trie 树的 3 个基本性质。

- 根节点不包含字符，除根节点外每一个节点都包含且只包含一个字符。
- 从根节点到某一节点，路径上经过的字符连接起来，为该节点对应的字符串。
- 每个节点的所有子节点包含的字符都各不相同。

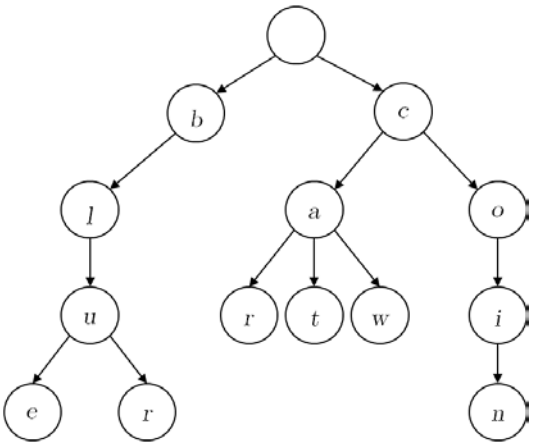


图 9-28　Trie 树

Trie 树可以用来构造字典，适合所有元素都是由字母和数字标记的情形，它为诸如字符串等的查找提供了一种高效的方案和途径。

为了查找方便，可以约定 Trie 树中同一层兄弟节点之间是存在有序关系的。例如，假设节点中存储的是字母，那么就可以按照字母表的顺序将同一层兄弟节点从左向右排列。在此基础之上可以使用树的孩子兄弟表示法来表示一个 Trie 树，其中每个节点应当包括 3 个域。首先是存储关键码的值域，其次是存储指向第 1 棵子树根节点的长子节点指针域，最后是存储指向兄弟节点的兄弟节点指针域。有的教材称使用树的孩子兄弟表示法表示的一个 Trie 树为双链树，并将双链树和 Trie 树统称为键树。本书将它们都看作是 Trie 树。9.4.2 节将介绍基于多重链表表示的 Trie 树。

9.4.2 Trie 树的表示

采用树的多重链表表示法来存储 Trie 树是一种很不错的存储方式。如图 9-29 所示，采用一个多重链表对如图 9-28 所示的 Trie 树进行存储。树中每个节点包含d个指针域（d是树中节点的最大度）。此外，为了节省指针所占用的空间，对于从 Trie 树中某个节点到叶子的路径上每个节点都只有一个孩子的情况，可把该路径上所有节点压缩成一个“叶子节点”，且在该叶子节点中存储关键字及其他有效信息。因此，Trie 树中是存在两种节点的，即分支节点和叶子节点。其中分支节点是包含有d个指针域和一个指示该节点中非空指针域个数的整数域。而叶子节点则含有关键字域和指向记录的指针域。另外在分支节点中不设数据域，每个分支节点所表示的字符均由其双亲节点中指向该节点的指针所在位置决定。

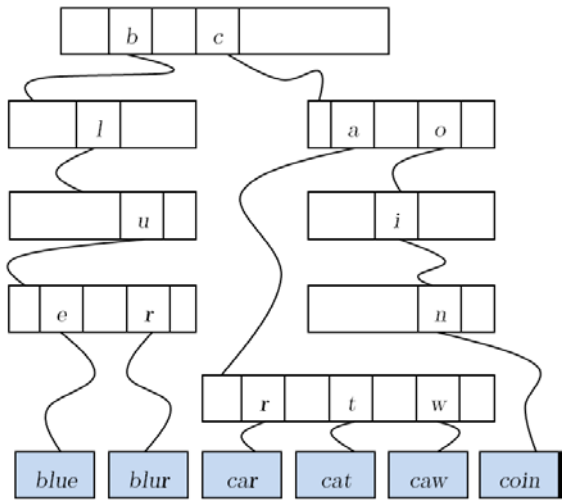


图 9-29　Trie 树的存储

易知，在 Trie 树上进行查找的过程就是从根节点出发，不断分叉和比较的过程。这个过程沿着给定值相应的指针逐层向下比对，直到叶子节点。如果叶子节点中的关键码和给定值相等，就表示查找成功。否则，如果分支节点中和给定值相应的指针为空，或叶节点中的关键字和给定值不相等，则表示查找不成功，即此 Trie 树中不存在待查找元素。

在 Trie 树上进行元素的插入和删除是比较容易的，因此这里不再赘言。9.4.3 节给出了

一个完整的 Trie 树类实现代码, 读者可以参阅其中相关成员函数的实现方法进行学习。

9.4.3　Trie 树的实现

本节给出 Trie 树类的实现代码。Trie 树的主要成员函数如下。

- search(const char* word) const;
- insert(const char* word);
- remove(const char* word);
- printall(char *pre, Trie_node *p);
- printpre(char *pre);

其中, search 函数用来查找是否一个前缀存在于 Trie 树中。insert 函数将一个前缀插入 Trie 树中。remove 函数用来删除一个前缀。printall 函数实现的功能是打印所有前缀, 它在执行时会对 Trie 树进行遍历。Printpre 函数打印符合前缀的单词。

```
#include <iostream>
#include <string>

using namespace std;

const int num_chars = 26;

class Trie {

protected:
    struct Trie_node {
        char* word;       //存储的字符
        Trie_node* branch[num_chars];
        Trie_node();
   };
   Trie_node* root;

public:
    Trie();
    int search(const char* word) const;     //查找字符
    int insert(const char* word);           //插入字符
    int remove(const char* word);
    void printall(char *pre, Trie_node *p); //打印所有字符
    void printpre(char *pre);               //打印符合前缀的字符
};

Trie::Trie():root(NULL)
{
}

Trie::Trie_node::Trie_node()
{
    word = NULL;
    for (int i=0; i<num_chars; ++i)         //重新释放 Tire 树
        branch[i] = NULL;
}
```

```
int Trie::search(const char* word) const
{
    char char_code;
    Trie_node *location = root;
    while( location!=NULL && *word!=0 )
    {//逐级查找标志位
        if (*word>='A' && *word<='Z')
            char_code = *word-'A';
        else if (*word>='a' && *word<='z')
            char_code = *word-'a';
        else
            return 0;
        location = location->branch[char_code];
        word++;
    }

    if (location != NULL && location->word != NULL)
        return 1;
    else
        return 0;         //没有找到
}

int Trie::insert(const char* word)
{
    cout<<"插入"<<word<<endl;
    int result = 1;
    if ( root == NULL ) root = new Trie_node;
    char char_code;
    Trie_node *location = root;

    while( location!=NULL && *word!=0)
    {   //通过标志位查找
        if (*word>='A' && *word<='Z')
            char_code = *word-'A';
        else if (*word>='a' && *word<='z')
            char_code = *word-'a';
        else
            return 0;     //没有找到合适位置，插入失败
        if( location->branch[char_code] == NULL )
            location->branch[char_code] = new Trie_node;
        location = location->branch[char_code];
        word++;
    }

    if (location->word != NULL)
        result = 0;
    else
    {
        location->word = new char[strlen(word)+1];
        strcpy(location->word, word);
    }
    return result;
}

int Trie::remove(const char* word)
{
    cout<<"删除"<<word<<endl;
    int result = 1;
```

```
    char char_code;
    Trie_node *location = root;

    while( location!=NULL && *word!=0)
    {
        if (*word>='A' && *word<='Z')
            char_code = *word-'A';
        else if (*word>='a' && *word<='z')
            char_code = *word-'a';
        else
            return 0;
        location = location->branch[char_code]; //删除前缀，修改标识
        word++;
    }

    if (location)
    {
        delete location->word;
        location->word = NULL;
    }
    else
        result = 0;  //删除失败

    return result;
}

//对树进行全部节点、前缀打印
void Trie::printall(char *pre, Trie_node *p)
{
    if (p->word != NULL) {
        cout<<pre<<endl;
    }

    for (int i=0; i<num_chars; ++i)
    {
        if (p->branch[i] != NULL)
        {
            char ch = 'a' + i;
            char *prestr = new char[strlen(pre)+2];
            sprintf(prestr, "%s%c", pre, ch);
            printall(prestr, p->branch[i]);
            delete [] prestr;     //释放内存
        }
    }
}

void Trie::printpre(char *pre)
{
   char char_code;
   char *p = pre;
   Trie_node *location = root;

   while( location!=NULL && *pre!=0 )
    {       //判断是否符合前缀
        if (*pre>='A' && *pre<='Z')
            char_code = *pre-'A';
        else if (*pre>='a' && *pre<='z')
            char_code = *pre-'a';
        else
```

```
            return;      //非法字符
        location = location->branch[char_code];
        pre++;
    }
    if (location != NULL)
        printall(p, location);
    return;
}
```

下面给出用于测试上述类的一个示例程序。该程序首先建立如图 9-28 所示的 Trie 树，然后对其中单词进行查找，最后输出前缀相同的一组单词。

```
void main()
{
    Trie t;
    t.insert("blue");
    t.insert("blur");
    t.insert("car");
    t.insert("cat");
    t.insert("caw");
    t.insert("coin");
    t.insert("coim");
    t.remove("coim");
    cout<<endl;

    cout<<"打印树中所有节点"<<endl;
    t.printpre("");
    cout<<endl;

    cout<<"开始查找"<<"cat"<<endl;
    if (t.search("cat"))
        cout<<"cat"<<" was found."<<endl;
    else
        cout<<"cat"<<" was not found."<<endl;

    cout<<"打印前缀为 ca 的单词:"<<endl;
    t.printpre("ca");
}
```

完成编码后编译并运行上述程序即可。

本章参考文献

[1] Georgy Adelson-Velsky, Evgenii M. Landis. An algorithm for the organization of informatio Z n. Proceedings of the USSR Academy of Sciences (in Russian), 146: 263~266. English translation by Myron J. Ricci in Soviet Math. Doklady, 3: 1259~1263, 1962.

[2] Rudolf Bayer. Symmetric Binary B-Trees: Data Structures and Maintenance Algorithms. Acta Informatica, 1: 290~306, 1972.

[3] Leo Guibas, Robert Sedgewick. A Dichromatic Framework for Balanced Trees. Proceedings of the 19th IEEE Symposia of Foundations of Computer Science: 8-21, 1978.

[4] Mark Allen Weiss. 数据结构与问题求解（C++版）. 北京 :清华大学出版社, 2004.

第 10 章

集合与字典——再言搜索之话题

集合是现代数学中一个非常重要的概念，它在某些情况下又被称为类、族或者搜索。本章主要介绍有关集合的一些概念和知识。作为一些元素的汇集，现实中的很多集合都非常庞大，且集合中元素与元素之间也不存在任何必然的关系，这给集合的搜索工作带来了很大的负担。为了更好地组织集合中的数据，提高元素的搜索效率，人们设计了字典这种结构。字典可以认为是一种特殊的集合，在理解了集合的有关概念基础之上，本章还将对字典和散列的有关内容进行介绍。

10.1 集合论基础

本节将对集合的概念和集合的基本运算进行简要介绍。

10.1.1 集合的概念

集合论公认的创始人是德国数学家格奥尔格·康托（Georg Cantor）。集合是集合论的核心概念。集合这个概念存在于我们日常生活的方方面面，下面列举几个集合的例子。

- 高三（6）班的全体同学。
- 所有三角形。
- 平面上满足方程$x^2 + y^2 < R^2$的所有点。
- 所有哺乳动物。

由以上例子可见，一个集合就是由若干（有限或者无限）成员构成的一个群集，它是能作为整体论述的事物的集体。集合中的每一个事物叫作集合的元素或成员。若某个元素存在于集合之中，则称这个集合包含此元素，或此元素属于该集合，用符号∈表示。如集合A包含元素b，则记为$b \in A$。若某个元素不存在于集合之中，则称这个集合不包含此元素，或此元素不属于该集合，用符号∉表示。如集合A不包含元素c，则记为$c \notin A$。

通常有 3 种表示集合的方式，分别为列举法、描述法及归纳定义法。限于本书所讨论的内容，在这里只对列举法及描述法进行介绍。有关归纳定义法的内容有兴趣的读者可以查阅其他相关资料。

下面首先介绍列举法。顾名思义，列举法就是把集合中的元素一个一个地列举出来。例如，中国古典四大名著可用下面的集合表示：

$$A = \{\text{西游记，三国演义，水浒传，红楼梦}\}$$

列举法能够清晰地表示出集合中的元素，使人一目了然。列举法还能比较容易地体现出集合中元素之间的规律。但用列举法表示的集合书写起来比较烦琐，而且列举法不利于对集合进行符号化表示，这给集合上的数学运算带来了一些不便。

若集合中元素的个数较多并且元素间的规律性较强，则可利用省略号来代替未被表示出的元素。如奇数集合可采用如下的形式表示：

$$O = \{\cdots -5, -3, -1, 1, 3, 5 \cdots\}$$

与列举法不同，描述法利用谓词描述出集合中元素的公共特征。例如1～35之间的整数集用描述法表示如下：

$$A = \{a \mid a \in \mathbb{Z} \wedge 1 \leqslant a \leqslant 35\}$$

描述法能够清晰地描述出集合中元素共同满足的条件并把集合符号化。用描述法表示的集合便于进行数学运算。但描述法不利于对集合进行形象化的表示。利用描述法表示的集合一般比较抽象，只有通过计算才能获得集合中的某个具体元素。

集合中的元素不仅可以是单独的事物，还可以是另一个集合。也就是说集合可以包含集合。如前面曾经举过的例子：高三（6）班的全体同学可以由两个集合组成，即男同学集合与女同学集合。同理，所有三角形也可以由两个集合组成，即直角三角形集合和非直角三角形集合。集合间的这种关系叫作集合间的包含关系。它的定义如下：设A和B是集合，如果A的每一个元素都是B的一个元素，那么A就是B的子集，记为$A \subseteq B$，读作“B包含A”或“A包含于B中”。若用谓词表达式表示集合间的包含关系可得：

$$A \subseteq B \Leftrightarrow \forall x\{x \in A \Rightarrow x \in B\}$$

如果$A \subseteq B$且$A \neq B$，那么称集合A是集合B的真子集，记作$A \subset B$，读作“B真包含A”，或“A真包含于B”。若$A \subseteq X$，则$A^C = \{a \mid a \in X, a \notin A\}$称为$A$的补集。

集合A的所有子集所组成的集合称为幂集，记为$\mathcal{P}(A)$。另外，以集合为元素的集合称为集族，记为$\mathcal{A}$。设$\mathcal{A}$是一个非空集族，$\mathcal{A}$的指标函数是从某一个集合$\mathcal{J}$到$\mathcal{A}$的一个满射f，其中$\mathcal{J}$被称为指标集，族$\mathcal{A}$连同指标函数f一起被称为一个集的加标族或加标集族。给定$\alpha \in \mathcal{J}$，集合$f(\alpha)$记成符号A_α。这个加标集族本身则记作$\{A_\alpha\}_{\alpha \in \mathcal{J}}$，可以读作“$\alpha$取遍$\mathcal{J}$时，所有$A_\alpha$的族”。当指标集自明时，则简单地记为$\{A_\alpha\}$。

在此之上，还可以定义覆盖和分划的概念。设$\mathcal{A} = \{A_\alpha\}_{\alpha \in \mathcal{J}}$是空间$X$的一个子集族，若：

$$X = \bigcup_{\alpha \in \mathcal{J}} A_\alpha$$

则称$\mathcal{A}$覆盖X，或称$\mathcal{A}$是X的一个覆盖。若$\mathcal{A}$的每一个成员都是X的开子集，则称它为X的

一个开覆盖。

同样设$\mathcal{A}=\{A_\alpha\}_{\alpha\in\mathcal{J}}$是空间$X$的一个子集族，且$\mathcal{A}$覆盖$X$。若$\alpha\neq\beta$，则$A_\alpha\cap A_\beta=\emptyset$。同时，$\forall\alpha\in\mathcal{J}$，$A_\alpha\neq\emptyset$。则称$\{A_\alpha\}_{\alpha\in\mathcal{J}}$为$X$的一个分划。

集合间的包含关系提供了一种判断两个集合是否相等的有效手段。判断方法如下：设A和B是集合，$A=B$当且仅当$A\subseteq B$且$B\subseteq A$。在后面的章节中可以看到“ == ”运算符正是利用上面的定理来判断两个集合是否相等的。

只含有一个元素的集合称为单元素集。注意单元素集与单元素是不同的，集合{1}是一个集合，这个集合中只有一个元素，这个元素是1。{1}与1不同，后者只是一个数字，并非集合。

单元素集中只有一个元素并不意味着在单元素集中只能出现一个事物。集合{{1,2}}同样是单元素集，因为集合中只有一个元素而且这个元素是另一个集合，即集合{1,2}。但这个单元素集中却出现了两个事物，分别为1和2。

不包含任何元素的集合叫作空集或者零集，并用符合∅表示。注意集合{∅}不是空集，它是一个单元素集，只有一个集合类元素∅。由集合间的包含关系可知，空集是任意集合的子集，即对于任意集合A均有$\emptyset\subseteq A$。集合A中元素的个数称为集合的势，记为$|A|$。

由于利用计算机表示的任何事物均是对真实事物的一种近似，加之计算机内部的资源有限，不可能用它来表示无限的事物，所以本章所讨论的集合均为有限集。也就是说存在一个集合U，计算机中表示的任何集合均是集合U的一个子集，即对任意集合A，均有$A\subseteq U$。

计算机中的集合与数学中的集合虽有相似之处，但二者之间仍然存在区别。这种区别主要表现在两个方面。首先数学中的集合可以是无限集，即集合中的元素可以有无穷多个。但是计算机中存储的数据必然是有限的，因此计算机中的集合都是有限集。其次，由集合的定义可知集合中的元素是没有排列顺序的。但为了方便运算，计算机存储的集合中元素是有序排放的。为了达到这一目地，集合中必须存放相同数据类型的元素，并且需要自定义元素之间的关系运算符“ < ”或“ > ”，根据这个运算符的结果可以对集合中的元素进行排序。这种存放有序元素的集合称为有序集。读者要注意把握数学中集合与计算机中集合之间的区别与联系，合理地利用程序对计算机中的集合进行操作。不可将两者等价视之，也不可把一些数学中有关集合的定理不假思索地应用到计算机所表示的集合中。

10.1.2　集合的运算

集合与集合之间可以进行一定的运算。本节将介绍几种常见的集合间的运算。这些运算在后续章节中均有各自的实现部分。

- 并运算

集合间的并运算的规则可表述为：设A、B是集合，A和B的并记为$A\cup B$，且$A\cup B$也

是集合，并有：

$$A \cup B = \{x | x \in A \vee x \in B\}$$

例如，设有$A = \{1, 2, 3, 4, 5\}$，$B = \{2, 3, 5, 7\}$，则$A \cup B = \{1, 2, 3, 4, 5, 7\}$。如图 10-1 所示，图中阴影表示$A \cup B$。

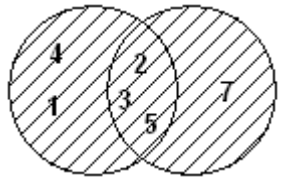


图 10-1　$A \cup B$

集合的并运算是满足交换律及结合律的，即$A \cup B = B \cup A$，$(A \cup B) \cup C = A \cup (B \cup C)$。

- 交运算

集合间的交运算的规则可表述为：设A、B是集合，A和B的交记为$A \cap B$，且$A \cap B$也是集合，并有：

$$A \cap B = \{x | x \in A \wedge x \in B\}$$

例如，若$A = \{1, 2, 3, 4, 5\}$，$B = \{2, 3, 5, 7\}$，则$A \cap B = \{2, 3, 5\}$。如图 10-2 所示，图中阴影表示$A \cap B$。

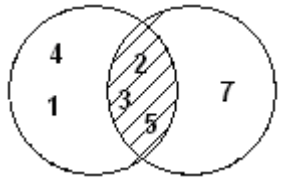


图 10-2　$A \cap B$

集合的交运算同样满足交换律及结合率，即$A \cap B = B \cap A$，$(A \cap B) \cap C = A \cap (B \cap C)$。

- 差运算

集合间的差运算的规则可表述为：设A、B是集合，A和B的差记为$A - B$，且$A - B$也是集合，并有：

$$A - B = \{x | x \in A \wedge x \notin B\}$$

例如，若$A = \{1, 2, 3, 4, 5\}$，$B = \{2, 3, 5, 7\}$，则$A - B = \{1, 4\}$。如图 10-3 所示，图中阴影表示$A - B$。

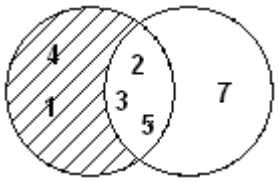


图 10-3　$A-B$

注意，集合的差运算并不满足交换律与结合律。

本章中为了编程方便，规定运算符“ $*$ ”代表集合之间的交运算，运算符“ $+$ ”代表集合之间的并运算，运算符“ $-$ ”代表集合之间的差运算。由于编译系统并不知道以上运算符的新含义，所以在使用运算符$*$、$+$、$-$之前要对它们进行重载。

10.2　集合的实现

本节主要介绍计算机中集合的表现形式及其实现方法。在计算机中，集合的实现方法很多，我们将介绍其中较为常用的位向量表示法和链表表示法。

10.2.1　位向量集合

什么是位向量呢？请看下面的例子。

电影院中有一排预留的座位，规定如果不是座位的主人，其他人是不能占用这个预定座位的。如果一位顾客已经预定了座位，那么他只能在他预定的座位上观看电影，不可以放弃预定的座位或临时更换座位。假设第 1 个座位由A预定，第 2 个座位由B预定，C没有预定座位，只是一个普通的观众。那么除非A亲自来观看电影，否则谁也没有权力抢占A所预定的位置，无论是B还是C。如果A来看电影，那么他只能去坐他所预定的座位（1号座位），他既不可以与B，也不可以与C调换座位。基于以上规定，电影开场后哪个预定座位的顾客没来看电影便一目了然：谁的座位是空着的，谁一定没来看电影。

上述例子便应用了位向量的思想。如果用1表示座位中有顾客，用0表示座位中没有顾客，那么由一串 0、1 的序列便可得知预留座位中顾客上座的情况。

下面再来看一个例子，这个例子说明了位向量在计算机中的应用方式。图 10-4 为 Pentium 系列微处理器标志寄存器前12位示意图。

标志寄存器的作用是标识程序执行的状态。以上12位的标志寄存器就是利用位向量实现的。程序当前的运行状态由位向量的每一个二进制位进行标识。当某个二进制位存储数字1时，表示程序执行过程中出现了此标志位所标识的状态，CPU 将识别此状态并进行相应的处理；当某个二进制位存储数字0时，表示程序执行过程中并没有出现此标志位所标识的状态，CPU 转而处理其他状态。

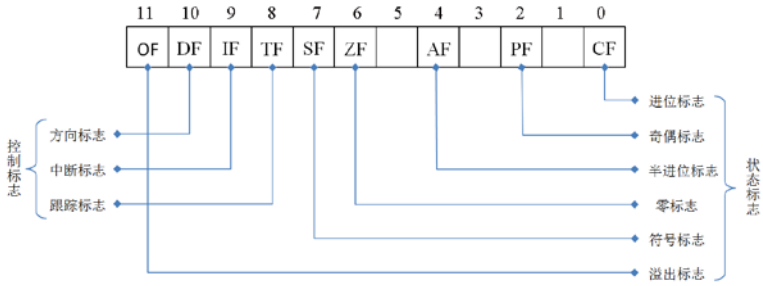


图 10-4　标志寄存器示意图

从以上两个例子中不难看出位向量其实是一个数组，数组中每一个元素都是一个二进制位（0 或者 1）。位向量中并不存储实际的数据而只存储数据的状态，并且这个状态必须具有二值性。例如，某个数据是否出现，某个数据是否已被使用。若某个数据具有 3 个或 3 个以上的状态，则这个数据的状态不能使用位向量的一位来表示。

计算机中能表示的数据是有限的，不可能用计算机来表示无限的数据。计算机中每种数据都有一定的类型，每种类型的数据都只有有限个。这就从理论上确保了使用位向量表示数据状态的可行性。如果一个位向量取得足够大，使得计算机中每种类型数据的每个状态均能在这个位向量中找到对应的二进制位，那么这个位向量就能够表示计算机中的所有数据。如果把计算机中的所有数据组成一个集合（前文已提到这个集合为U），那么利用这个位向量就能够把集合U充分地描述出来。

在实际应用中位向量不必取得如此之大，只需根据情况选择适当长度的位向量即可。太长的位向量不仅不利于管理，还会白白地浪费宝贵的存储资源。

用位向量表示集合就是把位向量中的每一位与全集中的每一个元素对应起来。若某个元素包含于某个集合，那么便把位向量中表示这个元素的二进制位置为1；若某个元素不包含于某个集合，那么便把位向量中表示这个元素的二进制位置为0。图 10-5 给出了一个使用位向量来表示集合的例子。

$A=\{1,2,3,4,5\}$

1	2	3	4	5	7
1	1	1	1	1	0

$B=\{2,\ 3,\ 5,7\}$

1	2	3	4	5	7
0	1	1	0	1	1

图 10-5　集合的位向量表示法

图 10-5 中分别给出了集合A与集合B的位向量表示形式，由图可知位向量中每个元素与全集U中每个元素成一一对应关系，位向量的每一种状态分别对应全局U的某一个子集。若对集合A与集合B做“并”运算，则所得集合$A \cup B$及其位向量如图 10-6 所示。

$$A \cup B = \{1, 2, 3, 4, 5, 7\}$$

1	2	3	4	5	7
1	1	1	1	1	1

图 10-6　$A \cup B$及其位向量示意图

可见集合$A \cup B$的位向量等于集合A的位向量与集合B的位向量在对应位置上进行或运算所得的结果。在这个例子中$A \cup B$等于全集U，所以$A \cup B$的位向量每一位均为1。

若对集合A与集合B做“ 交 ”运算，则所得集合$A \cap B$与$A \cap B$的位向量如图 10-7 所示。图 10-7。

$$A \cap B = \{2, 3, 5\}$$

1	2	3	4	5	7
0	1	1	0	1	0

图 10-7　$A \cap B$及其位向量示意图

可见集合$A \cap B$的位向量等于集合A的位向量与集合B的位向量在对应位置上进行与运算所得的结果。若互不相交的两个集合进行与运算，由于结果为空集$\emptyset$，所以其对应的位向量中每一个二进制位均为0。

若对集合A与集合B做“ 差 ”运算，则所得集合$A - B$与$A - B$的位向量如图 10-8 所示。

$$A - B = \{1, 4\}$$

1	2	3	4	5	7
1	0	0	1	0	0

图 10-8　$A - B$及其位向量示意图

集合A的位向量与集合B的位向量计算得到$A - B$位向量的运算规则可归结为表 10-1。

表 10-1　集合位向量差运算规则

A	1	1	0	0
B	1	0	1	0
$A - B$	0	1	0	0

下面给出位向量集合 VecSet 的声明部分。

```
const int DefaultSize = 100;
class VecSet{
    int* contain;
    int size;
public:
    VecSet(int MaxSize = DefaultSize);
    ~VecSet();

    void Add(int add);
    void Del(int del);
    void MakeEmpty();
```

```
    int GetSize(){return size;}
    bool IsMember(const int x);
    VecSet& operator+(VecSet& another);
    VecSet& operator-(VecSet& another);
    VecSet& operator*(VecSet& another);
    VecSet& operator=(VecSet& another);
    bool operator==(VecSet& another);
    friend ostream& operator<<(ostream& stream,VecSet& set);
};
```

类 VecSet 包含两个私有数据成员 contain 与 size。其中，contain 是指向位向量数组首地址的指针，size 指定了位向量的长度。由于位向量数组利用二进制数0与1实现对变量状态的记录，所以这个数组是整型的数组。

带一个参数的构造函数 VecSet(int MaxSize)建立一个长度为 MaxSize 的位向量，在默认情况下，我们为 MaxSize 赋一个默认值 DefaultSize。

函数 Add(int add)用于向位向量中添加数 add，即把代表 add 的二进制位置为1。函数 Del(int del)用于从位向量中删除数 del，即把代表 del 的二进制位置为0。运算符“+”用于求两个集合的并运算。运算符“-”用于求两个集合的差。运算符“*”用于求两个集合的交。运算符“=”用于把集合 another 整体赋给当前集合。运算符“==”用于判断两个集合是否相等。运算符“<<”用于输出当前集合。

下面给出类 VecSet 的实现部分。此外，本书的在线支持资源中另提供有一个该类的测试程序，读者可自行下载后用于验证该类的使用。

```
VecSet::~VecSet()
{
    if(contain != NULL)
        delete [] contain;
}

VecSet::VecSet(int MaxSize):size(MaxSize){
    assert(MaxSize>0);
    contain = new int[size];
    assert(contain != NULL);
    MakeEmpty();
}

void VecSet::Add(int add){
    assert(add>=0 && add<size);
    contain[add]=1;
}

void VecSet::Del(int del){
    assert(del>=0 && del<size);
    contain[del]=0;
}

void VecSet::MakeEmpty(){
    for(int i=0;i<size;i++)
        contain[i]=0;
}

bool VecSet::IsMember(const int x)
{
```

```
    assert(x >= 0 && x < size);
    return contain[x] != 0;
}

VecSet& VecSet::operator+(VecSet& another){
    assert(this->GetSize() == another.GetSize());
    VecSet* temp = new VecSet(this->GetSize());
    for(int i=0;i<size;i++){
        (*temp).contain[i] = contain[i]||another.contain[i];
    }
    return *temp;
}

VecSet& VecSet::operator-(VecSet& another){
    assert(this->GetSize() == another.GetSize());
    VecSet* temp = new VecSet(this->GetSize());
    for(int i=0;i<size;i++){
        (*temp).contain[i] = contain[i]&&(!another.contain[i]);
    }
    return *temp;
}

VecSet& VecSet::operator*(VecSet& another){
    assert(this->GetSize() == another.GetSize());
    VecSet* temp = new VecSet(this->GetSize());
    for(int i=0;i<size;i++){
        (*temp).contain[i] = contain[i]&&another.contain[i];
    }
    return *temp;
}

VecSet& VecSet::operator=(VecSet& another){
    assert(this->GetSize() == another.GetSize());
    for(int i=0;i<size;i++){
        contain[i] = another.contain[i];
    }
    return *this;
}

bool VecSet::operator == (VecSet& another){
    assert(this->GetSize() == another.GetSize());
    for(int i=0;i<size;i++){
        if(contain[i] != another.contain[i])
            return false;
    }
    return true;
}

ostream& operator<<(ostream& stream,VecSet& set){
    for(int i=0;i<set.GetSize();i++){
        if(set.IsMember(i)) cout<<i<<" ";
    }
    cout<<endl;
    return stream;
}
```

以上函数的实现部分均比较简单，在这里不做过多的解释。但请注意操作符“－”的实现部分。逻辑运算符&&的运算规则是当&&左边的条件为假时，运算结果总为假。当且仅当运算符&&左边的二进制值为1，且右边的二进制值为0时运算结果为1。所以

函数体使用了一个运算符“！”来辅助实现这种运算规则。

类 VecSet 中使用的位向量比较简单。若初始化位向量的长度为10，则这个位向量记录从0～9之间10个整数在集合中的出现状态；若初始化位向量的长度为20，则这个位向量记录从0～19之间20个整数在集合中的出现状态。概括地说，类 VecSet 实现的集合是一个整数集，这个整数集所属的全集U中有n个整数，范围是从 0～$n-1$，n的取值由构造函数的参数 MaxSize 指定。由此可见集合 VecSet 的应用范围是很有限的，它最多只能存储从0～$n-1$之间n个连续的整数。

在实际应用中读者可根据需求自行创建集合类 VecSet，使类 VecSet 中能够存放不同类型的数据；还可自行构建位向量，使位向量能够记录不同类型数据在集合中的出现状态。创建新类与构建新位向量的过程可仿照上述代码。

在创建过程中应注意n的取值。若n的值取得过大，位向量要占用过多的存储空间，这无疑是对存储资源的一种浪费；若n的值取得过小，那么集合 VecSet 所属的全集U便随之减小，很有可能把用户需要用到的数据排除在外。总之，n的取值要根据实际精确把握，既要确保位向量不占用过多的内存空间，又要确保全集U中含有足够多的元素。

位向量集合的优点是实现方式简单，算法时间复杂度较低。由于利用静态数组实现，所以向集合中添加元素与从集合中删除元素的操作均具有接近常数的时间复杂度。这意味着即使集合中元素很多，程序也能相当快速地向集合中添加或删除元素。

位向量集合的另一个优点是元素与元素之间没有关联，添加或删除某个元素不会对集合中的其他元素造成任何影响。这一方面使得元素的添加与删除操作的实现异常简单，另一方面使得位向量的不同状态数达到最大。例如，若某个位向量占据15个比特位，那么由这个位向量表示的集合便有2^{15}个之多。

鉴于以上种种优点，当实际应用中数据不是很多，且数据量的变动不大时常采用位向量的方法实现集合。

10.2.2 单链表集合

在介绍链表集合的概念及其实现之前，先来看一个例子。

某公司现要统计一种新产品在市场上的使用情况。该公司决定把消费者按年龄的大小分为三等，分别为青年、中年及老年，并分别统计各个年龄段的消费者购买该公司产品的人数及消费者的个人信息。为了完成这个任务，该公司建立了 3 个集合，用于存储 3 个年龄段消费者的个人信息。现在的问题是，由于事前不可能做出准确的估计，这 3 个集合究竟取多大才合适呢？

一种解决方案是把 3 个集合均开辟得足够大，使它们能存储下足够多的消费者信息。在这种实现方式中可以利用位向量集合，但必须在事前把所有消费者的个人信息输入公司的数据库中。当某个人购买了该公司的产品，公司便把集合中代表这个人的二进

制位置为1，并在数据库中读取有关这个人的个人信息。据统计，该地区现有大约10万消费者，若利用这种调查方式，则该公司事前要向数据库中存储10万人的个人信息！这显然是不符合实际情况的。这不仅浪费了大量的存储资源，还会大大降低调查工作的速度。

由以上例子可以看出集合的位向量表示法虽然有种种优点，但在实际应用中还是存在一定的不足和缺陷。由于位向量集合使用静态数组，所以它不能动态地调整大小。当不能对集合中元素的个数做出准确估计时，只能把位向量集合定义得足够大。位向量集合的另一个不足之处是操作起来比较烦琐。位向量集合中不存储实际的元素，而只存储代表实际元素状态的二进制位。所以当需要对实际的元素进行访问时，程序需要不断地对数据源发出请求，这样降低了工作效率。

如何解决以上问题，使得集合既能动态地调整大小，又能存储实际的元素？本节介绍的链表集合就是一种十分有效的解决方案。

链表集合就是用链表实现的集合。从实现方式来看，链表集合本质上就是一个特殊的链表，在这个链表中元素有序地存储，而且元素之间不存在重复，这就对链表的插入与删除方式提出了很高的要求。由此可见用于实现链表集合的链表不同于一般意义上的链表。

从功能上来看，链表集合能够完成有关集合的任何操作，包括集合间的并、交、差等运算。由于在功能上具有完备性，用户在使用链表集合时可不必关心集合的底层实现方式，而把链表集合完全当作是纯粹的集合使用。这就要求实现链表集合的算法具有良好的接口。

由于链表类中存储的是实际元素而不是代表元素状态的二进制位，所以为了达到支持用户自定义数据类型的目的，类 ListSet 要采用模板机制。采用模板机制可使类 ListSet 具有更强的通用性，更优异的可移植性，也使得类 ListSet 更接近于现实中的集合。

利用模板机制设计的类 ListSet 存在一个问题。链表集合其实是一个特殊的链表，在这个链表中元素有序地排放且元素之间不能出现重复。既然链表中元素的排放是有序的，所以元素之间一定存在着大小关系。用户在利用类 ListSet 存储自定义的数据类型时一定要在自定义的数据类型中重载运算符“ < ”或“ > ”。程序正是利用逻辑运算符“ < ”或“ > ”实现对元素的有序排放。

现实生活中有很多事物并不能比较大小，比如用类 ListSet 实现一个水果的集合，在这个集合中橘子是存放在苹果之前还是之后？再如利用类 ListSet 实现一个有关等腰三角形的集合，究竟什么样的等腰三角形是“ 大 ”的？什么样的等腰三角形是“ 小 ”的？在这个集合中哪个等腰三角形应该排在最前头？可见利用类 ListSet 实现的集合也只是对数学中集合的一种近似，它与数学中的集合存在一定的差别，而且某些场合下这种差别可能相当大。

在上述有关水果的集合中可利用水果名称的拼音字母来为某种水果排序。如此一来，

橘子便排在苹果前头，也就是说橘子“小于”苹果。但在有关等腰三角形的集合中就没有办法应用同样的策略对之进行排序。这里可以按照三角形的面积确定三角形之间的大小关系，但由于集合中元素的不可重复性，具有相同面积的等腰三角形便被视为是同一个三角形；也可以利用三角形顶角的大小来为三角形排序，但这样具有相同顶角的三角形便被集合视为是同一个三角形。如何确定元素之间的大小关系是应用类 ListSet 存储用户自定义类型数据的关键。有兴趣的读者可以试着思考一下，利用什么样的原则才能够有效地对等腰三角形进行“排序”。

下面给出类 ListSet 的定义部分。

```
template <class T> class ListSet;                          //类的前视声明

template <class T>
class ListSetNode
{
    friend class ListSet<T>;                               //声明友元类
    T data;
    ListSetNode<T>* link;
public:
    ListSetNode() :link(NULL){}
    ListSetNode(T value) :link(NULL), data(value){}
};

template <class T>
class ListSet
{
    ListSetNode<T>* head;
    ListSetNode<T>* tail;
public:
    ListSet();
    ~ListSet();
    ListSet(ListSet & lset);                               //复制构造函数

    void Add(T add);
    void Del(T del);
    bool IsEmpty();
    void MakeEmpty();
    ListSet<T>& operator+(ListSet<T>& another);
    ListSet<T>& operator-(ListSet<T>& another);
    ListSet<T>& operator*(ListSet<T>& another);
    ListSet<T>& operator=(ListSet<T>& another);
    bool operator==(ListSet<T>& another);
    void Output();
    ListSetNode<T>* GetHead(){ return head; }
};
```

由于链表集合利用链表实现，所以在对类 ListSet 进行定义之前先要定义节点类 ListSetNode。节点类 ListSetNode 中有两个数据成员，分别为节点中存储的数据 data 与指向后继的指针 link。由于考虑代码的封装性，类 ListSetNode 被设计成为类 ListSet 的私有成员，所以在定义类 ListSetNode 时要声明类 ListSet 为该类的友元类，以方便类 ListSet 对类 ListSetNode 中私有数据的操作。类 ListSetNode 默认的构造函数用于创建一个新节点，该节点中存储一个系统自动分配的数据 data，并且指针 link 指向 NULL。在实际应用中常用该类的第 2 个构造函数，这个构造函数接收一个参数 value。value

的值即是该节点中存储的 data 值。这个节点指向后继的指针 link 仍然为 NULL。

与类 ListSetNode 类似，类 ListSet 中也有两个私有数据成员，分别为链表头指针 head 与尾指针 tail。指针 head 指向链表的表头节点，它标识了链表集合的开始；指针 tail 指向链表的表尾节点，它标识了链表集合的结束。由于该类中成员函数比较多，且功能复杂，为了表述方便，下面分别讲解类中各成员函数的实现部分。首先给出 ListSet 类中构造函数与析构函数的实现。

```
template <class T>
ListSet<T>::ListSet()
{
    this->head = new ListSetNode<T>();
    this->tail = this->head;
}

template <class T>
ListSet<T>::~ListSet()
{
    MakeEmpty();
    delete head;
}

template <class T>
ListSet<T>::ListSet(ListSet<T>& lset)
{
    this->head = new ListSetNode<T>();
    this->tail = this->head;
    ListSetNode<T>* current = lset.head->link;
    while (current != NULL)
    {
        Add(current->data);
        current = current->link;
    }
}
```

上述代码中的构造函数之功能是创建只有一个表头节点的空链表集合。引入表头节点是为了统一对空集与非空集的操作。调用该函数后集合与链表中元素的状态如图 10-9 所示。

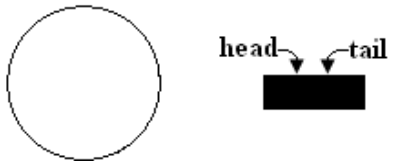


图 10-9　创建空集合

向集合中添加元素的函数实现如下。

```
template <class T>
void ListSet<T>::Add(T value)
{
    ListSetNode<T>* add = new ListSetNode<T>(value);
    ListSetNode<T>* preCurrent = head;              //记录当前节点
    ListSetNode<T>* current = preCurrent->link;     //记录当前节点的前驱节点
```

```
    while (current != NULL && current->data < value)//寻找插入位置
    {
        preCurrent = current;
        current = preCurrent->link;
    }
    if (head == tail && current == NULL)            //向空链表中插入节点
    {
        head->link = add;
        tail = add;
    }
    if (head != tail && current == NULL)            //向链表尾插入值 add
    {
        preCurrent->link = add;
        add->link = NULL;
        tail = add;
    }
    if (current != NULL && current->data == value)  //链表中已存在值 add
    {
        return;
    }
    if (current != NULL && current->data > value)
    {
        preCurrent->link = add;
        add->link = current;
    }
}
```

由于集合中元素具有唯一性及有序性，所以类 ListSet 中的函数 Add(T value)要比链表类 List 中的函数 Add(T value)复杂得多。当调用函数 Add(T value)时，程序首先检测当前链表集合是否为空。若为空集，直接将新元素添加到集合中并更新链表的尾指针；当集合不是空集时算法从表头节点开始遍历整个链表。如果链表当前节点中的元素小于元素 value，继续向链表尾部遍历；如果链表当前节点中的元素大于元素 value，则把 value 插到链表中当前节点之前，使 value 成为链表中当前节点的前驱；若完成对链表的遍历后仍找不到大于 value 的节点，此时把 value 直接插入到链表尾部即可。在遍历链表的过程中要注意记录链表中当前节点中 data 的值，一旦发现 data 的值与 value 相等，则说明链表中已经存在值 value。由集合中元素的唯一性，此时忽略添加元素的操作，算法直接结束。

向集合中添加元素 1、2、3、4、3、5 后，集合与链表中元素的状态如图 10-10 所示。

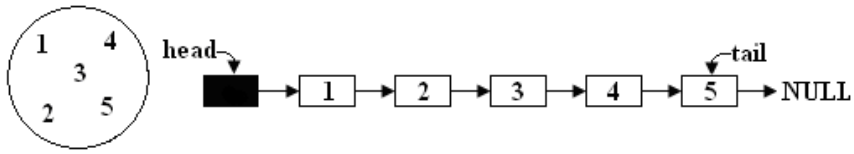


图 10-10　链表集合的添加过程

用于删除集合中某元素的函数实现如下。

```
template <class T>
void ListSet<T>::Del(T del)
{
    ListSetNode<T>* preCurrent = head;
```

```
    ListSetNode<T>* current = preCurrent->link;
    while (current != NULL && current->data != del)      //寻找删除节点
    {
        preCurrent = current;
        current = preCurrent->link;
    }
    if (current != NULL)
    {
        preCurrent->link = current->link;
        if (current->link == NULL){ tail = preCurrent; }//更新表尾指针
        delete current;
    }
}
```

函数 Del(T del)的实现部分比较简单。当执行它时算法首先遍历链表，当某个节点中的值与参数 del 相等时，算法把这个节点从链表中删除。当删除的节点位于链表尾部时，删除后要更新表尾指针 tail。被删除的节点要利用 delete 语句进行释放，以免发生内存泄漏。若欲删除的元素不存在于当前集合中，则算法什么也不做，自动“跳过”这个元素。删除集合中元素 4 的过程如图 10-11 所示。

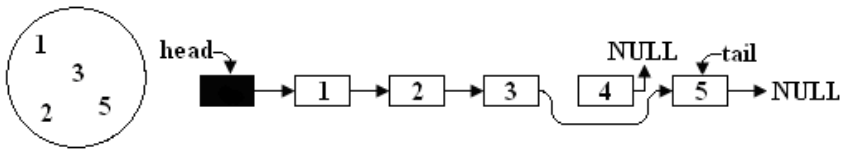


图 10-11　链表集合的删除过程

为了使用户方便使用，集合间的并、交、差等运算均采用运算符重载形式。这样做的好处是：首先，使链表集合间的运算更接近于数学上集合间的运算；其次，方便用户利用链表集合类。重载的运算符会使代码更加清晰易懂，使代码更加人性化。

运算符“＋”用于实现集合间的并运算，它的实现方式如下。

```
template <class T>
ListSet<T>& ListSet<T>::operator+(ListSet<T>& another)
{
    ListSet<T>* tmpSet = new ListSet(another);
    ListSetNode<T>* current = this->head->link;
    while (current != NULL)
    {
        tmpSet->Add(current->data);
        current = current->link;
    }
    return *tmpSet;                              //返回当前集合的引用
}
```

运算符“－”用于实现集合间的差运算。它的实现方式如下。

```
template <class T>
ListSet<T>& ListSet<T>::operator-(ListSet<T>& another)
{
    ListSet<T>* tmpSet = new ListSet(*this);
    ListSetNode<T>* current = another.head->link;
    while (current != NULL)
    {
```

```
        tmpSet->Del(current->data);
        current = current->link;
    }
    return *tmpSet;
}
```

运算符“*”用于实现集合间的交运算。它的实现方式如下。

```
template <class T>
ListSet<T>& ListSet<T>::operator*(ListSet<T>& another)
{
    ListSet<T>* tmpSet = new ListSet(*this);
    *tmpSet = *this - another;
    *tmpSet = *this - *tmpSet;
    return *tmpSet;
}
```

运算符“*”的实现部分十分简单。设有集合A与集合B，不难推出$A \cap B = A - (A - B)$。运算符“*”的实现便利用了该公式。

下面给出 ListSet 类中其余各成员函数的实现代码。本书的在线支持资源中另提供有一个该类的测试程序，读者可自行下载后用其验证该类中函数的使用。

```
template <class T>
bool ListSet<T>::IsEmpty()
{
    return head == tail;                          //判断集合是否为空
}

template <class T>
void ListSet<T>::MakeEmpty()
{
    ListSetNode<T>* current = head->link;
    ListSetNode<T>* del = NULL;
    while (head->link != NULL)                    //循环删除集合中的元素
    {
        head->link = current->link;
        del = current;
        current = current->link;
        delete del;                               //调用 delete 释放当前节点
    }
    tail = head;
}

template <class T>
ListSet<T>& ListSet<T>::operator=(ListSet<T>& another)
{
    MakeEmpty();
    ListSetNode<T>* current = another.head->link;
    while (current != NULL)
    {
        Add(current->data);
        current = current->link;
    }
    return *this;
}

template <class T>
bool ListSet<T>::operator==(ListSet<T>& another)
```

```
{
    ListSetNode<T>* current1 = head->link;
    ListSetNode<T>* current2 = another.head->link;
    while (current1 != NULL)
    {
        if (current2 == NULL){ return false; }
        if (current1->data != current2->data){ return false; }
        current1 = current1->link;
        current2 = current2->link;
    }
    if (current2 != NULL){ return false; }
    return true;
}

template <class T>
void ListSet<T>::Output()
{
    ListSetNode<T>* current = this->GetHead()->link;
    while(current != NULL){
        cout << current->data << " ";
        current = current->link;
    }
    cout << endl;
}
```

虽然类 ListSet 与类 VecSet 的接口部分十分相似，但由于两者的实现方式完全不同（类 ListSet 利用链表实现，而类 VecSet 利用静态数组实现），所以接口的实现部分相差很大。究竟哪一种实现方式更好？在实际应用中应该如何在两种集合中进行选择？

由于类 VecSet 采用静态数组实现，所以向类 VecSet 表示的集合中添加新元素或删除已有元素可较快完成。在向类 ListSet 表示的集合中添加新元素时，算法要不断比较链表当前节点数据与待加入数据的大小，以确保元素在集合中按照顺序排放；而向类 VecSet 表示的集合中添加新元素只需将位向量中代表待加元素的二进制位置为1即可，无须关心元素在集合中的排放顺序。所以在实际应用中当数据量不大，且数据量的变化亦不大时，利用位向量法可更有效地实现集合类。

在实际应用中当数据量很大，或数据量变化较大时，利用位向量实现的集合便不能有效地解决问题。因为位向量集合不能根据实际情况灵活地调整大小。这时可以应用链表集合类 ListSet。该类可根据实际情况灵活地调整大小，而且其对内存的使用是很“ 节约 ”的。若存储空间允许，类 ListSet 可以存储“ 无限 ”多的数据。类 ListSet 的缺点是该类中方法的时间复杂度较高，实现效率不如类 VecSet。

10.3　字典

集合主要用于存储数据，但仅仅把数据存储起来并不能满足实际应用的需求。如果一个仓库只是用作货物存放而无法对货物进行有效的提取，那么这个仓库就毫无实用价值。本节要介绍的字典便是一种既能存储数据，又能对数据进行有效提取的结构。字典与集合的关系十分密切，字典应用集合的理论进行元素的存放，但与集合不同的是，字典可以实现元素的高效搜索。

10.3.1 字典的概念

相信读者一定不会对字典感到陌生。现实生活中有很多字典，如新华字典、英汉大词典、成语词典等。通常意义的字典就是一种收集词汇或单字并将其按某种检字法（如音序检字法、部首笔画检字法等）进行排序，同时对相应的字词加注读音、释义、用法等以供人检索参考之用的工具书。

但生活中除了以上常用字典外，还有一类用于检索的工具书或小手册也可以被认为是一种字典，例如，列车时刻表、电话号码簿等。可见，此处所讲的字典已不再是一般意义上的字典。

在计算机科学中，字典也是一种数据结构类型，它也被称作映射或关联数组。数据结构中的字典是指一种集合，这种集合中每个元素由两部分组成，分别为关键码（也称为键值）和属性（也称为值），即字典就是一种〈关键码-属性〉对的集合。本书前面介绍过的 map 就是 STL 中用以实现关联数组的容器类型。

字典中包含关键码和属性的二元组称作关联。通常可以使用文件或表格来表示实际的对象集合，也就是将关联存进文件或表格中以备查阅。用文件中的记录或表格中的表项来表示单个对象。〈关键码-属性〉被存于记录（或表项）中，通过表项的关键码来标识该表项。这就相当于现实中字典的正文部分，词条及其对应释义是字典正文部分的基本单位，那么字典的正文部分也就是诸多这样基本单位的有序排列。以查阅一本英汉字典为例，读者可以根据生词的字母顺序查找其在词典中的特定位置，然后就可以找到该词的相关释义。

在上述过程中，用于定位的生词的某些字母称为这个词的关键码，词典中的解释称为这个词的属性。在某本电话号码簿中寻找某个单位的电话号码也遵循上述查字典的过程。首先要确定单位的名称，然后在电话号码簿中根据单位的名称寻找这个单位的电话。在这个过程中，单位名称就是关键码，单位的电话号码就是属性。

字典是如何将搜索变得简单的呢？同样以查阅一本英汉字典为例，如果读者希望查找单词“ Structure ”的意思，则应首先翻阅目录，查找到以 S 开头的单词所在的起始页码，然后翻到该页再继续按字母顺序进行查找。数据结构中的字典采用了同样的思想，字典表项的存放位置及其关键码之间的对应关系可以用关键码及表项位置指针这样一个二元组来表示。这个二元组构成了搜索某一指定表项的索引项。可以使用多种方式搜索索引项。但是使用散列结构是一种搜索效率很高的词典的组织方法。散列会在本章后续内容中介绍。

字典中任意两个不同的元素间要有不同的关键码，也就是说任一元素应该能由它的关键码唯一标识。例如，在英汉词典中每个单词均有唯一的拼写方式，不同单词间拼写方式一定不同；电话号码簿中每个单位均有专属自己的名字，不同单位的名字也一定不同。由此可见关键码与元素之间成一一映射关系。

建立字典最重要的环节就是关键码的选择。好的关键码不仅要能唯一标识元素，还应

当比较容易产生。比如现在要统计某大学在册本科生的个人信息，并把这些信息组织在一个字典中，如何选取关键码才能使上述工作较快完成？选择学生的性别作为关键码显然是不合理的，这样所有男同学便共用一个关键码，而所有女同学也共用一个关键码。选择学生出生日期作为关键码要比选择性别作为关键码稍好一些，可是同样会有一部分同学具有相同的生日，这部分同学之间将无法彼此区分。究竟怎样选取关键码才能唯一地标识每个学生？答案是利用学号。由于每个学生只有一个学号，而学号在学生入校时便已产生，所以学号作为关键码是可行而且方便的。

现实中并不是每个事物都能够产生唯一标识它的关键码，因为当事物的数量很大时关键码的产生过程可能相当烦琐。例如，现在要给某图书馆的馆藏书籍进行编号，若该图书馆有 1 亿册藏书，则每本书的关键码要长达 7 位。这么长的关键码不但不方便记忆，还会给图书的管理工作造成很多麻烦。

在这里可以使用多级关键码对图书进行编号。如第 1 级关键码指明图书的种类（如科技类、文学类、历史类、法律类等），第 2 级关键码指明图书所分属的学科（如科技类图书包括：数学、物理、化学、生物、计算机等），第 3 级关键码指明图书所在书架的号码。采用这种分级编码方式不仅使产生关键码的过程大大简化，还会使关键码更容易记忆，图书更容易管理。

数据结构中的字典是指以一定分类方法进行分类的一组数据的集合，一个班中注册学习任意一门课程的学生构成了一个字典。当有新的学生注册时，老师将学生信息加入班级中，当有人要放弃这门课程时，则删除此人的记录。在上课过程中，老师可以查询名册以得到与某特定学生相关的记录。这就告诉我们，作为一种数据结构，字典也有其相应的操作。与字典有关的基本操作包括：

- 插入具有给定关键码及对应属性的元素。
- 在字典中寻找具有给定关键码的元素。
- 从字典中删除具有给定关键码的元素。

实现字典一般都要借助于本书前面所介绍过的各种树形搜索结构，从而实现效率上的提升。例如，红黑树、AVL 树或者字典树都是常用的选择。鉴于我们已经对这些内容做过详尽的介绍，此处将不会使用它们来组织字典。为了演示字典结构的建立和使用，下面采用最简单的单链表来实现字典结构。特别地，作为范例下面的程序将建立一个英汉字典，即以英文单词作为键值，以汉语释义作为属性。下面首先给出词条类的源代码，它是在链表节点类的基础上修改而来的。

```
class Item {

    string english;
    string chinese;
    Item*  link;

public:

    Item() : link(NULL){}
```

```
    Item(string en, string ch) : link(NULL), english(en),chinese(ch) {}
    ~Item(){};
    void SetLink(Item* next);
    Item*  GetLink();
    string GetIndex();
    string GetValue();
};

void Item::SetLink(Item* next){
    link = next;
}

Item* Item::GetLink(){
    return link;
}

string Item::GetIndex(){//返回节点中的数据
    return english;
}

string Item::GetValue(){//返回节点中的数据
    return chinese;
}
```

接下来给出的是字典类实现的源代码。

```
class Dictionary{

    Item* head;
    Item* tail;
public:
    Dictionary();
    virtual ~Dictionary();

    bool Add(string en, string ch);
    bool Del(string en);
    string Search(string en);
    int GetCount();
    void RemoveAll();
};

Dictionary::Dictionary()
{
    head = new Item();
    tail = head;
    tail->SetLink(NULL);
}

Dictionary::~Dictionary()
{
    RemoveAll();
    delete head;
}

bool Dictionary ::Add(string en, string ch){
    Item* add = new Item(en, ch);
    tail->SetLink(add);
    tail = tail->GetLink();
    tail->SetLink(NULL);
    if (tail != NULL)
```

```
        return true;
    else
        return false;
}

bool Dictionary::Del(string en){

    Item* cur, *curPre;
    cur = head;
    curPre = cur->GetLink();

    while (cur != NULL){
        if (curPre->GetIndex() == en)
            break;
        cur = cur->GetLink();
        if (cur !=NULL)
            curPre = curPre->GetLink();
    }

    if (tail == curPre)
        tail = cur;
    cur->SetLink(curPre->GetLink());  //将被删除节点从链表中摘下
    delete curPre;

    if (curPre != NULL)
        return true;
    else
        return false;
}

string  Dictionary::Search(string en) {

    Item* cur;
    cur = head->GetLink();
    while (cur != NULL){
        if (en == cur->GetIndex())
            break;
        cur = cur->GetLink();
    }
    if (cur != NULL)
        return cur->GetValue(); //返回节点中的 value
    else
        return "Cannot find";
}

int Dictionary::GetCount() {
    int count = 0;
    Item* current = head->GetLink();
    while (current != NULL) {
        ++count;
        current = current->GetLink();  //顺序移动 cur
    }
    return count;
}

void Dictionary::RemoveAll() { //删除所有节点
    Item* cur;
    while (head->GetLink() != NULL) {
        cur = head->GetLink();
        head->SetLink(cur->GetLink());
```

```
            delete cur;
        }
        tail = head; //置表尾于表头处
}
```

本书的在线支持资源中另提供有一个该类的测试程序，读者可下载后用于验证该类中各函数的使用。可见该类利用线性的链表结构简单地实现了关联数组的功能，包括增加〈关键码-属性〉对，根据关键码搜索属性，以及依据关键码删除条目等功能。那么该程序有没有什么不足呢？首先，在关联数组中，关键码应保持唯一性。在这一点上，我们的程序并未做相应处理。显然，在增加〈关键码-属性〉对时，如果能够先判别一下相应的关键码是否已经存在便可解决该问题。另外，以上字典中条目的存储是乱序的，而且搜索也是线性的。这会导致性能的下降，特别是当字典的规模相当庞大时。前面已经讲过，实际应用中，字典的组织往往要借助于非线性的搜索树来完成。而每添加一个〈关键码-属性〉对，树形结构也势必要进行调整（例如，为新节点寻找合适的插入位置，以及保持树的平衡性等）。这两部分的功能留给读者思考，如果能独立完善上述两方面的功能也就表明读者的工夫已经到家了。

10.3.2 搜索运算

当元素按照一定的编码方式产生自己的关键码后，这个关键码便和元素的属性一同被存储到字典中。此时，元素的关键码与元素的属性组成了一个〈关键码-属性〉二元组，字典中存储的便是许许多多这样的二元组。

字典不但要存储数据，更要方便用户对数据进行访问。查字典的过程其实就是用户对字典中元素进行访问的过程。好字典应该使用户能够快速地找到某个〈关键码-属性〉二元组，从而降低用户的等待时间。

本节将介绍几种字典中常见的搜索运算，这些运算方式可有效地降低搜索所花费的平均时间，从而提高用户使用字典的效率。

搜索运算从总体上可分为静态搜索与动态搜索。静态搜索是在静态环境中进行的。所谓的静态环境是指搜索结构在执行插入与删除等操作后不发生改变。这种方式实现方法简单，操作方便，但算法效率不高，且容易发生溢出等问题。

动态搜索是在动态环境中进行的。所谓的动态环境是指搜索结构在执行插入与删除等操作时可自动调整自身状态，以保证每次搜索运算均可具有较高的效率。这种方式可以大大减少搜索时间、提高算法执行效率，而且不易发生溢出错误。但这种搜索方法实现起来比较复杂。

下面举例说明静态搜索与动态搜索之间的联系与差别。

在乘坐火车之前火车站的工作人员要对乘客进行检查，以确保没有乘客携带易燃易爆品上车。出于安全方面的考虑，来火车站乘坐火车的每名乘客均需要接受检查。这样，当乘客很多时检查过程就进行得相对慢些，当乘客较少时检查过程就进行得相对快些。这个过程可以看作对易燃易爆品进行静态搜索，搜索结构是乘车旅客排成的长队。新

乘客到来时只能自动站到队列末尾处，此时前面旅客已经排好的队列并没有被改变。这就是前面所说的静态搜索不改变搜索结构。这个例子显示出了静态搜索大致的工作模式。

考虑在图书馆中的借书过程。首先读者要查明欲借图书的馆藏号，然后带着这个馆藏号去相应藏书室寻找图书。由于图书的馆藏号中记录了存放图书的书架号，所以即使藏书室很大，读者也能快速找到某一本特定的图书。如果在寻找图书的过程中图书馆陆续有新书上架，那么新到的书籍可能会打乱图书馆原有书籍的排列顺序。但由于原书的馆藏号不变，即使图书排列顺序发生变化，读者还是可以很快找到欲借阅的书籍。这个过程可以看作是对某本书进行动态搜索，即动态搜索改变原搜索结构。

本书前面已经对动态搜索做过诸多详细的介绍。本节将只介绍静态搜索的概念及实现方法。静态搜索中比较具有代表性的方式是顺序搜索与折半搜索。下面将分别介绍这两种搜索方法的工作原理及实现过程。

1．顺序搜索

顺序搜索是一种简单、普遍的搜索方法。上述火车站安检的例子就是顺序搜索在现实世界中的应用。顺序搜索从搜索结构的前端开始，顺序用各对象的关键码与给定的关键码进行比较，若关键码匹配则说明搜索成功，此时返回元素的值或元素的引用；若整个搜索结构都已检测完而仍未发现与给定关键码匹配的元素，则搜索失败，此时返回失败信息。

下面给出顺序搜索类 OrderSearch 的定义部分。

```
template <class T>
class Item{
    int key;
    T data;
public:
    Item():key(-1){}
    Item(int keyInput,T value):key(keyInput),data(value){assert(key>=0);}
    int GetKey(){return key;}
    T GetData(){return data;}
    friend ostream& operator<<(ostream& stream,Item<T>& item);
};

template <class T>
class OrderSearch{
    Item<T>* contain;
    int size;
public:
    OrderSearch():contain(NULL),size(0){}
    OrderSearch(int maxSize);
    void Add(Item<T> add);
    int GetSize(){return size;}
    Item<T>* GetContain(){return contain;}
    Item<T>& Search(int k);
    friend ostream& operator<<(ostream& stream,OrderSearch<T>& set);
};
```

由于字典中存储的是〈关键码-属性〉二元组，所以在定义类 OrderSearch 之前先要定

义二元组类 Item。类 Item 有两个数据成员，分别为关键码 key 与属性 data。由于这两个数据成员均为私有类型，所以要定义函数 GetKey()与 GetData()，分别用来返回当前二元组中数据成员 key 与 data 的值。

类 Item 中重载的流运算符用于输出当前二元组中的数据，其实现代码如下。代码的实现部分利用了前面提到的函数 GetKey()与 GetData()。函数的实现比较简单，这里不再做过多的解释。

```
template <class T>
ostream& operator<<(ostream& stream,Item<T>& item){
    cout<<"key:"<<item.GetKey()<<"\t"<<"data:"<<item.GetData()<<endl;
    return stream;
}
```

类 OrderSearch 中定义了两个私有成员变量 contain 与 size。其中变量 contain 是一个指向静态数组的指针，这个静态数组用于存放二元组类 Item 的实例。变量 size 记录字典中最多可以存放的二元组数目，即数组 contain 的长度。

类 OrderSearch 有两个构造函数。不带参数的构造函数用于建立一个空的字典，该字典中不存放任何二元组 ;带一个参数的构造函数用于建立一个大小为 maxSize 的字典，该字典中最多存放的二元组数目由参数 maxSize 指定。

下面给出以上函数的实现部分。

```
template <class T>
OrderSearch<T>::OrderSearch(int maxSize)
{
    contain = new Item<T>[maxSize];                  //建立数组，存放二元组
    size = maxSize;                                  //记录数组大小
    for(int i=0;i<maxSize;i++)                       //初始化字典
    {
        Item<T> ini;
        contain[i] = ini;
    }
}

template <class T>
void OrderSearch<T>::Add(Item<T> add)
{
    assert(add.GetKey()>=0 && add.GetKey()<size);    //判断 key 的合法性
        contain[add.GetKey()] = add;                 //添加二元组
}

template <class T>
Item<T>& OrderSearch<T>::Search(int k)
{
    assert(k>=0 && k<size);                          //判断 key 的合法性
    return contain[k];                               //返回二元组
}

template <class T>
ostream& operator<<(ostream& stream,OrderSearch<T>& set)
{
    for(int i=0;i<set.GetSize();i++)
```

```
    {
        if(set.GetContain()[i].GetKey() != -1)
        {
            cout<<"key:"<<set.GetContain()[i].GetKey()<<"\t";
            cout<<"data:"<<set.GetContain()[i].GetData()<<endl;
        }
    }
    return stream;
}
```

顺序搜索对搜索结构的要求很低，元素在搜索结构中可任意排列。这与接下来要介绍的折半搜索是不同的。折半搜索要求元素在搜索结构中有序排列。

顺序搜索最大的缺点是这种方法仅适用于字典较小的情况。算法每次对字典中所有元素进行遍历，直到找出满足要求的元素为止。由此可见，顺序搜索是种非常“ 不稳定 ”的搜索方法。当字典中第 1 个元素恰好是搜索的目标元素，显然顺序搜索利用很少的时间便能找到它。但当字典中最后一个元素才是搜索的目标元素时，算法只能遍历整个字典，直到把最后一个元素返回。当字典中存储数以百万记的元素时，利用顺序搜索的方法可能需要相当长的时间才能找到目标元素。顺序搜索算法的时间复杂度为$O(n)$。

2．折半搜索

与顺序搜索相比，折半搜索是一种更加快速的搜索方法。这种方法对集合中元素的个数没有限制，即使是很大的集合，折半搜索也能迅速地找出特定元素。

折半搜索虽然对集合中元素的个数没有限制，但它要求元素按照一定次序排列。下面将结合图示介绍折半搜索的工作原理。

假设现有集合A如图 10-12 所示。

3	5	11	17	24	28	36	40	45

图 10-12　集合 A

为了表述方便，集合A中只记录元素的关键码，而且这些关键码按照从小到大的顺序排列。现要应用折半搜索法搜索关键码为40的元素。首先算法用关键码40与集合中位于正中间的关键码（本例中为24）比较，如图 10-13 所示。这时，算法检测到位于集合正中间的关键码为24。由于此处元素按照从小到大的顺序排放，所以关键码为40的元素定位于关键码为24的元素之“ 右 ”。算法放弃集合的左半部，转而在集合的右半部中进行搜索。

3	5	11	17	24	28	36	40	45

图 10-13　第 1 次比较

在右半集合中的搜索方式与上述搜索方式相同。首先，算法用关键码40与右半集合中

位于正中间的关键码（此处为36）进行比较，如图 10-14 所示。

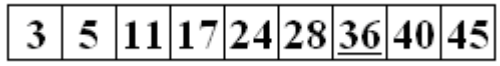


图 10-14　第 2 次比较

这时，算法检测到此位置中元素的关键码为36。由于40大于36，所以这时算法忽略子集合的左半部，转而在子集合的右半部中进行搜索。搜索过程与上述过程类似。经过 3 次比较，算法最终找到关键码为40的元素在集合中的位置。由上述过程可见，虽然集合中有9个元素，但算法只用了3次比较就找到了目标元素。若对以上集合应用顺序搜索，则需要进行8次比较。由此可见折半搜索算法的效率是很高的。折半搜索算法的时间复杂度为$O(\log_2 n)$。

本书前面曾经给出过折半搜索的递归实现，下面给出折半搜索算法的迭代实现。

```
template <class T>
int BinarySearch(List<T>& dic, T& item) {

    int lowKey = 0;
    int highKey = dic.GetCount() - 1;
    int midKey;

    while (lowKey <= highKey)
    {
        midKey = (lowKey + highKey) / 2;                //寻找中间节点
        if (dic.GetAt(midKey) < item)
        {
            lowKey = midKey + 1;                        //在右半集合中搜索
       }
        else if (dic.GetAt(midKey) > item)
        {
            highKey = midKey - 1;                       //在左半集合中搜索
        }
        else
            return midKey;
    }
    return -1;
}
```

10.4　散列

字典中表项的存放位置及其关键码之间的对应关系可以通过〈关键码-表项位置指针〉这个二元组来表示，该二元组构成了搜索某一指定表项的索引项。实际中，可以用来作为索引项搜索方式的选择有很多种，例如，前面介绍过的顺序查找法和折半查找法等。现实中有时待处理的数据量往往大得惊人，单纯从查找算法中寻找提高效率的途径是不够的，这时就需要找到一种高效的字典组织方法。目前，使用散列就是一种搜索效率很高的词典组织方法。本节介绍有关散列的一些问题。

10.4.1　散列的概念

散列又称为哈希或杂凑。散列方法在表项的存储位置与它的关键码之间建立一个确定的对应函数关系 Hash()，以使每个关键码与结构中的唯一存储位置相对应，该关系可用下式表示：

$$\text{Address} = \text{Hash}(\text{Record.key})$$

可见在存放表项时，通过相同函数计算存储位置，并按此位置存放，这种方法称为散列方法。基于散列方法构建的数据结构称为散列表，散列表是一种支持数据的快速插入、删除及查找等操作的数据结构。散列表中存储的元素仍然是二元组，该二元组由关键码与属性组成。散列表的特别之处在于表中元素的排放顺序。散列表中的元素既不是毫无规律地排列，也不是按照升序或降序简单排列。散列表中每个元素的存储位置都是由一个叫作散列函数的公式决定的，也就是说散列方法中使用的函数就是所谓的散列函数。

因为通常来说，关键码集合都要比散列表地址集合大得多，所以有可能经过散列函数的计算，把不同的关键码映射到同一个散列地址上。我们称这些散列地址相同的不同关键码为同义词。为了便于读者理解，下面用一个现实当中的例子来说明散列表及散列函数的意义。

有一种游戏机叫作弹子机，相信很多人都玩过。它的工作原理是这样的：在机器上方有一个小孔，玩家把弹子从这个小孔投入弹子机中。由于重力原因，被投入的弹子将在机器中做自由落体运动。弹子在下落过程中要不断与弹子机中的障碍物进行碰撞，这些碰撞会改变弹子的运动轨迹。弹子机底部有一些凹槽，每个凹槽均对应一定的奖赏。当弹子落在其中某一凹槽内时，玩家便赢得相应的奖赏。由于机器中障碍物的阻拦，弹子最终落在哪个凹槽完全是随机的。

上述过程便是对散列表工作方式的简单模拟。在这个过程中，弹子代表字典中某一个二元组，机器底部的凹槽代表字典，而机器中的障碍物便是散列函数。由此可见，经过散列函数的运算后，每个元素均可得到其在字典中的特定位置。

但弹子机与散列表间存在一个明显的区别：弹子机中每个弹子最终会落在哪一个凹槽中完全是随机的；而散列表中每个弹子在经过散列函数的运算后将存储在字典中的固定位置。

好的散列函数应该使元素在散列表中呈现均匀存储，而不是“堆叠”在某一个特定区域中。弹子机中的散列函数不是一个好的散列函数，因为它会使弹子集中分布在奖赏金额较小的凹槽里。这样的散列函数不但不能充分利用字典中的存储空间，还会给元素的搜索制造很大麻烦。这也是散列方法需要研究的一个主要问题，即对于给定的一个关键码集合，选择一个计算简单且地址分布比较均匀的散列函数，避免或尽量减少冲突。当然，由于关键码集合比散列表地址集合大，因此冲突在所难免，于是散列方法需要研究的另一个主要问题就是拟定解决冲突的方案。

前面所说的存储二元组的散列表其实可以认为是一种狭义的散列，这种散列又称为哈希图（Hash maps）。事实上，散列表有两种不同的形式，即哈希图和哈希集（Hash sets）。哈希图是储存〈键值-属性〉对的关联结构，哈希集则是一个用来记录集合中项目成员的结构。哈希集并不会在键值和属性之间产生映射。哈希集仅仅存储键值集合，这时可以认为哈希集中键值对应的属性为空。在本章后面的一个拼写错误检查程序中使用了哈希集。其中，一个用来记录易拼错单词的词汇表被存储在哈希集中。这时词汇表中没有存储〈键值-属性〉对，每个单词都是作为一个键值来使用的，其对应的属性是不存在的。哈希集可以用来回答诸如某个单词是否位于一个词表中这样一个问题。

10.4.2 散列函数

算法

最基本的 5 种散列算法包括直接定址法、除留余数法、平方取中法、乘余取整法、折叠法。

键值提供了对数据的间接引用。可以用一个函数将键值映射到一定的整数范围内，然后用整数值来访问数据，这种函数被称为散列函数或哈希函数。通常，构造散列函数应注意以下几个问题：首先，散列函数的定义域必须包括需要存储的全部关键码，而如果散列表允许有m个地址，其值域必须在0~$m-1$之间；其次，散列函数计算出来的地址应能均匀分布在整个地址空间中，从而减少冲突；再次，散列函数应当是尽量简单的。下面就来介绍几种常用的散列函数。

1．直接定址法

直接定址法利用元素关键码的某个线性函数值作为该元素的散列地址。散列地址即是元素最终在字典中的存储位置。如下面的函数式（其中a、b为常数）：

$$\text{Hash(key)} = a \times \text{key} + b$$

采用该方法，当向字典中加入某一新元素时算法便自动调用此函数以确定该元素最终的存储位置。若某元素关键码key为1，上式中a等于2，b等于3，则该元素最终存储在字典的第5个位置中。

直接定址法的优点是实现方式简单，算法时间复杂度较小，而且不会产生冲突。但直接定址法要求散列地址空间的大小与关键码集合的大小一致，而这种要求是很苛刻的，一般很难实现。例如，当关键码的范围为 1~1000000 时，元素散列地址的个数也要达到10^6。这么大的散列地址是不合实际的。

2．除留余数法

设散列表中允许的地址数为m，取一个不大于m，但最接近或等于m的质数k，或选取一个不含有小于20的质因子的合数作为除数。利用下面的式子计算元素的散列地址的方法称为除留余数法：

$$\text{Hash(key)} = \text{key}\%k, \qquad k \leqslant m$$

其中，“ % ”是整数除法取余的运算，要求这时的质数p不是接近2的幂。例如，当元素的关键码key等于2008，散列地址总数为50，这时取$k = 47$，则散列地址为：

$$\text{Hash}(2008) = 2008\%47 = 34$$

所以元素将存储在字典的第34个位置中。

除留余数法将有效缩减散列地址空间的大小，例如，上例散列地址空间中只有50个有效的散列地址。除留余数法的缺点是极易发生冲突。如关键码为1914的元素经过上述散列函数计算后也将获得散列地址34。此时出现的两个不同元素争用同一存储地址的情况就称为冲突。健全的算法应能尽量合理地解决冲突。有关冲突的解决方式及其实现方法将放在后文中讨论。

3．平方取中法

平方取中法是一种常用的实现散列函数的方法。最早提出平方取中法思想的人是冯·诺伊曼。1946 年，冯·诺伊曼和华伦泰·巴格曼（Valentine Bargmann）、迪恩·蒙哥马利（Deane Montgomery）合作，向海军武器实验室提交了报告《高阶线性系统求解》，对线性方程组的各种解法进行了系统的阐述，并探讨了利用计算机进行求解的可能性。这期间冯·诺伊曼提出了可以运用到当时计算机上的伪随机数生成方法，就是现在要讲的平方取中法。

在介绍平方取中法的工作方式之前先介绍内码的概念。

内码是整机文字系统中使用的二进制字符编码，是沟通输入、输出与系统平台之间的交换码。利用内码可以使文本的传输更加高效，并使文本更具通用性。例如，字母A的内码用八进制数表示为01，字母B的内码用八进制数表示为02。

平方取中法是一种先放大再集合的构造方法，这种构造模式先通过求关键字的平方值扩大相近数的差别，然后根据表长度取中间的几位数作为散列函数值，这种取中间数的方法是一种类随机方案，因此也可以认为平方取中法是一种产生伪随机数的方法。因为一个乘积的中间几位数和乘数的每一位都相关，所以由此产生的散列地址较为均匀。

利用平方取中法实现散列函数的过程如下：首先，利用一定的编码规则把元素的关键码转换成标识符。然后，求出标识符的内码表示，并计算内码的平方值。最后，取内码平方数的中间x位作为元素最终的散列地址。简而言之，即先计算构成关键码表示符的内码平方，然后按照散列表的大小取中间的若干位作为散列地址。

现举例说明，假如有一组关键字的内码为$(0100_8, 0110_8, 1010_8, 1001_8, 0111_8)$，那么根据平方取中法的计算规则首先将它们平方后得：

$$(0010000_8, 0012100_8, 1020100_8, 1002001_8, 0012321_8)$$

若取表长为1000，则可取中间的 3 位数作为散列地址集，于是得到最终的散列地址计算结果为$(100, 121, 201, 020, 123)$。

平方取中法中，地址空间内散列地址的数目一般为2的k次幂，并在计算出内码的平方后，根据k的大小决定最终散列地址的位数。如某个地址空间中散列地址的个数为128，

则最终取内码平方数中间7位作为元素最终的散列地址。

平方取中法是构造散列函数的常用方法，这种方法适合于关键码中每一位都有某些数字重复出现的情况。据研究表明，平方取中法产生的散列地址有较强的随机性。

4．乘余取整法

乘余取整法利用下面的式子计算元素的散列地址：

$$\text{Hash(key)} = \lfloor Z \times (a \times \text{key}\%1) \rfloor$$

其中，a是一个大于0小于1的常数。Z为一个整数。式子$a \times \text{key}\%1$表示对$a \times \text{key}$取小数部分，即$a \times \text{key}\%1 = a \times \text{key} - \lfloor a \times \text{key} \rfloor$。例如，当关键码为2008，小数$a$为0.6180339，整数$Z$为10000，则散列地址计算为：

$$\text{Hash}(2008) = \lfloor 10000 \times (0.6180339 \times 2008\%1) \rfloor = 120$$

乘余取整法不但会缩减散列地址空间的大小，还能极大减小冲突情况的发生概率。克努特对常数a的取法做了仔细的研究，发现虽然a取任何值都可以，但一般取黄金分割数0.6180339比较好。

5．折叠法

折叠法的工作方式很有趣。此方法把关键码从左至右划分为位数相等的几部分，每一部分的位数与散列地址位数相同。当关键码位数不能被散列地址位数整除时，最后一部分可取得短些。

折叠法有两种：移位法与分界法。移位法所采取的具体方式是把各部分的最后一位对齐相加。分界法所采取的具体方式是各部分不折断，而沿各部分的分界来回折叠，然后对齐相加，并将相加的结果当作散列地址。折叠法适用于关键码位数很多，且每一位上数字分布比较均匀的情况。

下面通过实例演示这两种方法的工作方式。设关键码key = 987654321。假设散列地址为4位，如图 10-15 所示即为移位法（左图）与分界法（右图）计算散列地址的示例。

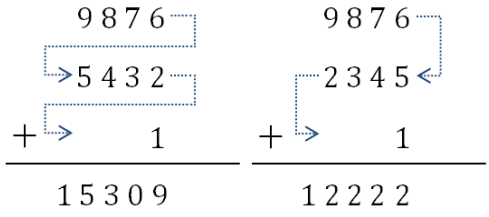


图 10-15　移位法与分界法

由式可见移位法计算结果为15309，由于散列地址为4位，所以舍去最高位数字1，元素最终的散列地址为5309。分界法计算结果为12222，同样舍去最高位数字1，元素最终的散列地址为2222。

10.4.3　字符串散列

算法

针对字符串散列，本节介绍包括 BKDR 散列、RS 散列和 FNV 散列在内的 3 种算法。

相比处理整数哈希而言，将字符串通过一个散列函数映射到地址表中要复杂许多。然而现实中这种需求却更加普遍。本节就来讨论几个专门为字符串设计的散列函数。

1．BKDR 散列

布莱恩·柯林汉（Brian Kernighan）和丹尼斯·里奇（Dennis Ritchie）在他们的经典著作《C 程序设计语言》一书中提出了这种散列算法。该算法通过一个循环进行计算，每次循环中，它将上一次循环中计算得到的结果值经过变换（即乘以31）后得到的新值同字符串当中当前字符的值相加。然后将该结果值同数组长度执行取模操作，其结果即作为函数的返回值。可见，该算法就是对字符串中每一个字符的 ASCII 码值进行的一个加和乘运算。原作者最初给出的乘数是31，后来也拓展到类似131, 1313, 13131…这样的数值。正如原作中所指出的，这并非是最好的散列函数，但却是比较简单有效的。

里奇是 C 语言和 UNIX 操作系统的发明人，他也因此与肯•汤普逊（Ken Thompson）一起荣获了 1983 年度的图灵奖。

下面我们给出该算法的实现代码，读者可借此加深对算法的认识。读者同时也应该注意到因为散列计算时采用的是无符号算术运算，因此保证了散列值非负。

```
unsigned long BKDRHash(const string& str)
{
    unsigned long seed = 31;
    unsigned long hashval = 0;

    for(int i = 0; i < str.length(); i++)
        hashval = hashval * seed + str[i];
    return hashval % HASHSIZE;
}
```

2．RS 散列

塞奇威克在他的著作《算法：C 语言实现》中提出了这种散列算法。他同时指出，这种算法仅用很小的额外开销就提供了随机的散列结果，对字符串关键字的散列有明显改进。该算法采用伪随机系数替代了固定基数，并且不使用预先计算出的一组随机数作为系数，而是在算法执行过程中不断产生这些系数。所以下面代码中的a和b也可以取其他值。例如，有的资料中令a = 63689，b = 378551。

```
unsigned long RSHash(const string& str)
{
    unsigned long a = 31415, b = 27183;
    unsigned long hashval = 0;
    for(int i = 0; i < str.length(); i++)
    {
        hashval = (hashval * a + str[i])%HASHSIZE;
        a = a * b % (HASHSIZE-1);
    }
    return hashval;
}
```

3．FNV 散列

FNV 散列算法，也称为 Fowler-Noll-Vo 算法，由格伦·福勒（Glenn Fowler）、兰

顿·诺尔（Landon Curt Noll）和方·沃（Phong Vo）于 1991 年左右共同提出。FNV 能快速地处理大量数据并保持较小的冲突率，它的高度分散性使它在处理一些非常相近的字符串时表现尤为突出。FNV 散列算法现在主要有两个版本，即 FNV-1 和 FNV-1a，算法的原作者推荐使用后者。

采用伪代码形式描述的 FNV-1 算法如下。

```
hash = offset_basis
for each octet_of_data to be hashed
    hash = hash * FNV_prime
    hash = hash xor octet_of_data
return hash
```

采用伪代码形式描述的 FNV-1a 算法如下。

```
hash = offset_basis
for each octet_of_data to be hashed
    hash = hash xor octet_of_data
    hash = hash * FNV_prime
return hash
```

与前面几种散列算法不同，FNV 散列会将结果映射到一个定长的大整数，通常以十六进制数表示。所以上述返回值 hash 就是一个n位的无符号整型哈希值，n通常取2的整数次幂，例如$32, 64, 128\cdots$这里的n位哈希结果指的是二进制位，所以如果n取32的话，则结果应该是一个8位的十六进制数，而且后面的参数选取都与n的大小有关。易见，FNV-1a 和 FNV-1 的唯一区别就是异或运算和乘法运算的顺序不同，它们所采用的参数都相同。循环体的处理过程是以哈希对象的每一个字节做单位的。

参数 offset_basis 是初始的哈希值，例如当$n = 32$时，offset_basis = 2166136261；当$n = 64$时，offset_basis = 14695981039346656037。其他情况的 offset_basis 取值可以在 FNV 的相关标准上查到，此处不一一列出。FNV_prime 是一个质数。此处同样仅给出它的前两种取值情况。当$n = 32$时，FNV_prime=16777619；当$n = 64$时，FNV_prime=1099511628211。这个质数的选取是该算法的关键。一方面不同的质数选择方案会对散列结果的优劣有影响，另一方面，算法执行过程中哈希值与质数相乘的结果，同时隐式地包含有取模的效果。例如当$n = 32$时，hash = hash * FNV_prime 的执行其实已同步完成了以2^{32}为除数取余的过程。此外，octet_of_data 是一个8位二进制数据（即 1 字节）。

下面给出32位 FNV-1a 散列算法的示例代码，FNV-1 算法可同理得到。

```
//const long offsetbasis32 = 2166136261;
#define FNV_32_INIT ((uint32_t)0x811c9dc5)
//const long FNVprime32 = 16777619;
#define FNV_32_PRIME ((uint32_t)0x01000193)

unsigned long FNV1a_32_Hash(const string& str)
{
    unsigned long hashval = FNV_32_INIT;
    for(int i = 0; i < str.length(); i++)
    {
        hashval = hashval ^ str[i];
```

```
        hashval = hashval * FNV_32_PRIME;
        //上式等价于下列位操作
        // hashval += (hashval <<1) + (hashval <<4)
        //           + (hashval <<7) + (hashval <<8) + (hashval <<24);
    }
    return hashval;
}
```

该段（不使用位操作的版本）代码仅仅是为了演示算法的执行。尽管它也可以得出正确的结果，但当哈希值的位数增加时，结果就会溢出，所以它仅适用于32位的情况。而且像上面那样执行大整数乘法效率也非常低。实际应用中的 FNV 散列函数都是基于位操作进行优化的，例如下面给出的64位 FNV-1a 散列算法之示例代码非但不会因为中间结果过大而溢出，而且具有很高的执行效率。但是这方面的内容已经超出了本书的范畴，因此这里不再对其执行过程做详细解释。

```
//const long offsetbasis64 = 14695981039346656037;
#define FNV_64_INIT ((uint64_t)0xcbf29ce484222325ULL);
//const long FNVprime64 = 1099511628211;
#define FNV_64_PRIME ((uint64_t)0x100000001b3ULL)

uint64_t FNV1a_64_Hash(const char* bp, size_t len)
{
    uint64_t hval = FNV_64_INIT;
    const char *be = bp + len;
    while (bp < be) {
        hval ^= (uint64_t) *bp++;
        hval += (hval << 1) + (hval << 4) + (hval << 5) +
                (hval << 7) + (hval << 8) + (hval << 40);
    }
    return hval;
}
```

本书的在线支持网站上另提供有此节所涉及之散列函数的测试程序，读者可自行下载后用于验证上述函数所执行之结果。

10.4.4　处理散列冲突

算法

常用于处理散列冲突的 3 种算法是线性探查法、二次探查法、双重散列法。

散列中经常会出现不同键值对应同一地址空间的现象，解决这种地址冲突的方法又称为溢出处理技术。在大部分常用的几种散列函数中冲突的产生是无法避免的，因此选择好的冲突解决方法十分重要。

处理冲突的办法从总体上说可分为两类：闭散列法与开散列法。闭散列法又称为开放定址法。开放定址法解决冲突的做法是：当冲突发生时，使用某种探查技术在散列表中形成一个探查序列。沿此序列逐个单元地查找，直到找到给定的关键字，或者碰到一个开放的地址（即该地址单元是空）为止。若要插入，在探查到开放的地址，则可将待插入的新节点存人该地址单元。查找时探查到开放的地址则表明表中无待查的关键字，即查找失败。开放地址法最常用的探查技术包括线性探查法与二次探查法等。下面来具体介绍这些探查方法。

1．线性探查法

求职面经点拨

请试回答附录题库中的第 30 题。

线性探查法的基本思想是将散列表$T[0\cdots m-1]$看成是一个循环向量，若初始探查的地址为d，即$h(\text{key})=d$，则最长的探查序列为$d,d+1,d+2,\cdots,m-1,0,1,\cdots,d-1$。探查时从地址$d$开始，先探查$T[d]$，然后依次探查$T[d+1]$，如此继续下去直到探查$T[m-1]$，此后又循环到$T[0],T[1],\cdots$直到探查到$T[d-1]$为止。在散列表中进行探查，如果对应位置已经被一个同义词占据，即发生冲突时，线性探查法依次向后寻找字典中没有存储元素的位置。直到发现一个空的单元时，就将元素插入。如果探查到$T[d-1]$时仍未发现空单元，则意味着插入失败。

现举例说明线性探查法的工作方式。假设有字典A，A中有10个位置，编号为0~9。若要向A中存储6个元素，这6个元素的关键码为$11,25,27,31,35,45$。散列函数为$\text{Hash(key)}=\text{key}\%10$。则经过哈希函数的计算后，6个元素在字典中的存储位置依次为$1,5,7,1,5,5$。可见第 1 个与第 4 个元素，第 2 个与第 5 个及第 6 个元素分别产生了冲突。当存储前 3 个元素时，由于未产生冲突，所以直接把它们加入字典即可，见图 10-16。

0	1	2	3	4	5	6	7	8	9
	11				25		27		

图 10-16　插入前 3 个元素

当插入第 4 个元素时，由于1号位置已被关键码为11的元素占据，根据线性探查法的工作原理，关键码为31的元素将被安排在2号位置，如图 10-17 所示。

0	1	2	3	4	5	6	7	8	9
	11	31			25		27		

图 10-17　插入第 4 个元素

与上述过程类似，关键码为35的元素将被安排在6号位置，如图 10-18 所示。

0	1	2	3	4	5	6	7	8	9
	11	31			25	35	27		

图 10-18　插入第 5 个元素

根据规则，关键码为45的元素应该存储在7号位置中，可是此时7号位置已被占用，所以算法继续向后探查，最后将把这个元素存储在8号位置中，如图 10-19 所示。

0	1	2	3	4	5	6	7	8	9
	11	31			25	35	27	45	

图 10-19　插入第 6 个元素

线性探查法实现过程比较简单，但算法容易使字典中的元素产生堆积。若向以上字典中存储关键码，分别为$1,11,21,31,41$，则这 5 个元素将被存进字典前 5 个位置，进而

导致字典的后半部分为空。下面介绍的二次探查法将有效地解决这个问题。

2．二次探查法

二次探查法的工作过程与线性探查法类似。在线性探查法中当元素间发生冲突时，算法将向后顺序搜索空位置，搜索步长为1。二次探查法不但可以向后顺序搜索，还可以向前逆序搜索，且搜索步长由搜索次数的平方决定。

下面举例说明二次探查法的工作过程。前文提到当向字典A中存储关键码分别为$1, 11, 21, 31, 41$的这 5 个元素时, 由于线性探查法的局部特性, 这 5 个元素将产生堆积。下面改用二次探查法添加这 5 个元素。

首先向集合中添加关键码为1的元素，如图 10-20 所示。

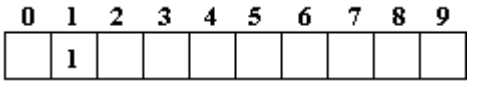


图 10-20　插入第 1 个元素

当添加关键码为11的元素时，字典产生冲突。此时算法按照search^2为步长在位置1前后寻找空位置，规定先向后寻找，再向前寻找。search的值为$1, 2, 3, \cdots, (m-1)/2$，其中m为字典的大小。按照以上原则，第 2 个元素将存储在位置2中，如图 10-21 所示。

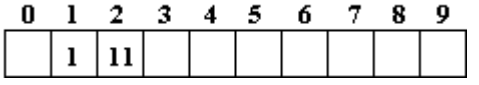


图 10-21　插入第 2 个元素

当添加关键码为21的元素时，发现1号位置的后继位置已被关键码为11的元素占据，此时算法将在1号位置的前驱中寻找空位置，步长仍为$1^2 = 1$，如图 10-22 所示。

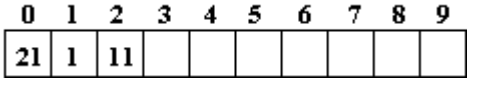


图 10-22　插入第 3 个元素

接下来添加关键码为31的元素。此时1号位置左右两边的位置均已被占用，于是算法改变步长，用新步长$2^2 = 4$重复上述查找过程。于是此元素将被安排在5号位置，如图 10-23 所示。

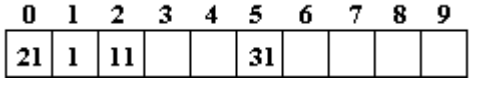


图 10-23　插入第 4 个元素

添加最后一个元素时，算法仍沿用步长4，但此时将在1号位置的左边查找空位置。到达字典头部后算法将返回字典尾部继续查找过程。最终字典中元素的状态如图 10-24 所示。

0	1	2	3	4	5	6	7	8	9
21	1	11			31		41		

图 10-24　添加第 5 个元素

5个元素全部添加完毕后字典中并没有产生堆积现象，由此可见二次探查法的确能够有效地避免堆积，从而使字典的存储空间得到更合理的利用。

3．双重散列法

该方法是开放定址法中最好的方法之一。它使用两个散列函数Hash()和ReHash()，故也称为双散列函数探查法。使用该方法时，第 1 个散列函数Hash()按表项的关键码key计算表项所在的位置。一旦发生冲突，则利用第 2 个散列函数ReHash()计算表项“下一个”位置的移位量。定义ReHash()的方法较多，但无论采用什么方法定义，都必须使ReHash()的值和m互为素数，这样才能使发生冲突的同义词地址均匀地分布在整个表中，否则可能造成同义词地址的循环计算。

闭散列法的一般处理方式都是将冲突的内容以一定模式存储在原地址中，发现冲突一般采用重新寻找的方法。与闭散列法相对的开散列法，又称为拉链法或链地址法，通常是将互为同义词的节点链成一个单链表，进行操作时查找链表。

具体操作时，拉链法将所有关键字为同义词的节点链接在同一个单链表中。若选定的散列表长度为m，则可将散列表定义为一个由m个头指针组成的指针数组$\boldsymbol{T}[m-1]$。凡是散列地址为i的节点，均插入到以$T[i]$为头指针的单链表中。T中各分量的初值均应为空指针。

与开放定址法相比，拉链法有如下几个优点。

- 拉链法处理冲突直观明了，且无堆积现象，解决了非同义词之间的冲突问题，平均查找长度及搜索复杂度有所降低。
- 由于拉链法中各链表上的节点空间是动态申请的，具有更好的扩展性。
- 在用拉链法构造的散列表中，删除节点的操作易于实现。只要删去链表上相应的节点即可。而对开放定址法构造的散列表，删除节点不能简单地将被删节点的空间置为空，否则将截断在它之后填人散列表的同义词节点的查找路径。这是因为各种开放定址法中，空地址单元都是查找失败的条件。因此在用开放定址法处理冲突的散列表上执行删除操作，只能在被删节点上做删除标记，而不能真正删除节点。

尽管拉链法相对于开放定址法有如上优点，但是当节点规模较小时选择线性探查法还是一个比较明智的选择，因为在这种情况下采用拉链法会浪费较多空间。

本节最后给出一道例题。希望能够通过该题目的求解过程加深读者对于散列函数及其冲突处理的理解。问题描述：设散列表为HT[13]，散列函数为$H(\text{key}) = \text{key}\%13$。用闭散列法解决冲突，对下列关键码序列$12, 23, 32, 18, 20, 03, 65, 44, 15, 49$造表。

（1）采用线性探查法寻找下一个空位，画出相应的散列表，并计算等概率下搜索成功的平均搜索长度和搜索不成功的平均搜索长度。

（2）采用双重散列法寻找下一个空位，散列函数为$\mathrm{RH(key)} = (7 * \mathrm{key})\%10 + 1$，寻找下一个空位的公式为$H_i = [H(\mathrm{key}) + \mathrm{RH(key)}]\%13$。画出相应的散列表，并计算等概率下搜索成功的平均搜索长度。

为了求解该题目，首先使用散列函数$H(\mathrm{key}) = \mathrm{key}\%13$求各项目键值得：

$$H(12) = 12 \quad H(23) = 10 \quad H(32) = 6 \quad H(18) = 5 \quad H(20) = 7$$
$$H(03) = 3 \quad H(65) = 0 \quad H(44) = 5 \quad H(15) = 2 \quad H(49) = 10$$

下面利用线性探查法造表。

0	1	2	3	4	5	6	7	8	9	10	11	12
65		15	03		18	32	20	44		23	49	12
(1)		(1)	(1)		(1)	(1)	(1)	(4)		(1)	(2)	(1)

搜索成功的平均搜索长度为：

$$\mathrm{ASL_{succ}} = \frac{1}{10}(1 + 1 + 1 + 1 + 1 + 1 + 4 + 1 + 2 + 1) = \frac{14}{10}$$

搜索不成功的平均搜索长度为：

$$\mathrm{ASL_{unsucc}} = \frac{1}{13}(2 + 1 + 3 + 2 + 1 + 5 + 4 + 3 + 2 + 1 + 5 + 4 + 3) = \frac{36}{13}$$

下面利用双重散列法造表，其中，$\mathrm{RH}(44) = 9$，$\mathrm{RH}(49) = 4$。所以有$H_{i(44)} = (5 + 9)\%13 = 1$，$H_{i(49)} = (10 + 4)\%13 = 1$。

0	1	2	3	4	5	6	7	8	9	10	11	12
65	44	15	03	49	18	32	20			23		12
(1)	(1)	(1)	(1)	(4)	(1)	(1)	(1)			(1)		(1)

此时，搜索成功的平均搜索长度为：

$$\mathrm{ASL_{succ}} = \frac{1}{10}(1 + 1 + 1 + 1 + 1 + 1 + 4 + 1 + 1 + 1) = \frac{13}{10}$$

散列的作用除了表现在数据组织和搜索上，在信息安全领域中散列发挥着更加重要的作用。随着互联网技术的发展，网络安全问题日益突出。散列除了提供高效的字典组织方式以外，在网络信息安全领域中也发挥着重要的作用。通信保密领域中的消息摘要算法就利用了散列函数：在消息摘要算法中，文件数据作为单向散列运算的输入，这个输入通过散列函数产生一个散列值。如果改动了文件，散列值就会相应地改变，接收者就可以检测到这种改动的痕迹。从理论上来讲，攻击者不可能自己造出一个消息来产生一个完全相同的摘要。散列函数可用于数字签名、消息的完整性检测、消息的起源认证检测等。在电子商务与电子政务中，信息的完整性被看作最重要和最基本的安全要求之一。散列函数作为一项重要的工具被广泛地应用于信息认证和数字签名中，成为保护信息完整性的主要手段。打个比方，散列函数的作用就是为信息产生一

个数字式指纹，即用一极短的数据串（散列值）来认证大量信息数据的完整性；以此确保接受方接收到的信息与发送方发送过来的信息一致，即在信息的传输过程中没有被第三方篡改、增减或重排。

经典问题求解

拼写检查问题。我们前面讲过，拼写检查一般是利用模糊匹配技术来实现的，读者能否看出此处我们所采取的方法其实是运用了之前介绍过的哪种模糊匹配技术的变型吗？

10.5 拼写检查问题

现在任何一种文字处理软件都会提供基本的拼写检查功能，例如我们常用的 Microsoft Office Word 等。使用过该软件的用户一定会发现，它不仅可以指出文章中可能存在的拼写错误，同时还可以提供可能的修正意见。下面就请读者来试着编写这样一个拼写检查程序。

本书提供了一个单词库，这个单词库辑录了生活中的很多常用英语单词，这里我们假设该词库包含了英语中的全部单词，单词库存放在 wordlist.txt 中，读者可以从本书的在线支持资源中下载得到这个文件，也可以自己编写一个词库。程序开始时首先载入这个文件。一旦程序发现该文件无法被正确载入则应输出相应的错误提示信息。如果该文件被顺利载入，那么程序将把其中的每个单词都存储到一个散列表中。接下来，程序打开一个文件并对其进行拼写检查。例如，程序可以读入如图 10-25 所示的这段文本，该文本存储在一个名为“ test.txt ”的文件中。

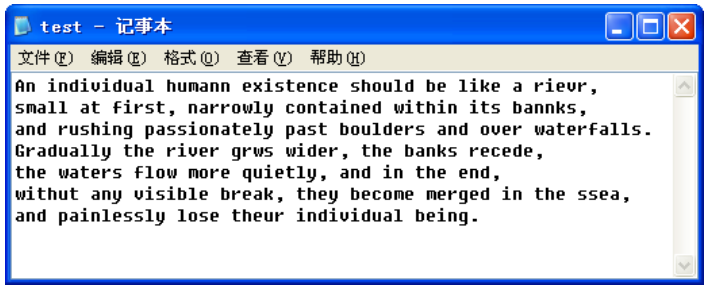


图 10-25 文本文件

打开文件后，程序会将文件中的每个单词与散列表中的单词进行比对。如果程序在散列表中不能找到文本文件中的某个单词，这时就认为该词是一个被错拼的单词。一旦程序发现错拼单词，程序应当提供一个单词表，该单词表提供了可能的修正意见。

为了提供每个错拼单词的修正意见，我们设计了一个简单的算法。这里假设每个错拼单词都是其本身的一个变形。这种所谓的变形将从如下 4 个方面来考虑。

- 删除任意字母

对于一个被错拼的单词，程序应当尝试删除其中某个字母，并查询这样是否能产生一个正确的单词。例如，对于错拼单词“ herre ”，删除其中一个字母 r，可能得到的正确单词是“ here ”。那么“ here ”就可以作为一个可能的修正意见。

- 任意字母被替换

对于一个被错拼的单词中的每个字符，程序都应该尝试将其替换为另外一个字母，并查询这样是否能产生一个正确的单词。例如，对于错拼单词“ hure ”，替换其中的字母 u，可能得到的正确单词有“ hare ”或者“ here ”等。那么这些都可以作为一个可能的修正意见。

- 置换相邻字母

对于一个被错拼的单词，程序应当尝试交换其中某两个相邻字母，并查询这样是否能产生一个正确的单词。例如，对于错拼单词“ heer ”，置换其中的相邻字母，可能得到的正确单词是“ here ”。那么“ here ”就可以作为一个可能的修正意见。

- 在单词的任意位置插入一个字母

对于一个被错拼的单词，程序应当尝试在其任意位置插入一个字母，并查询这样是否能产生一个正确的单词。例如，对于错拼单词“ hee ”，在其中插入一个字母 r，可能得到的正确单词是“ here ”。那么“ here ”就可以作为一个可能的修正意见。

基于上述原则就可以很容易地写出错拼词汇修正意见的算法了。

程序通过对散列表进行检索，并恰当地应用前面设计的算法对上述文本进行拼写检查，最终程序的输出如图 10-26 所示。

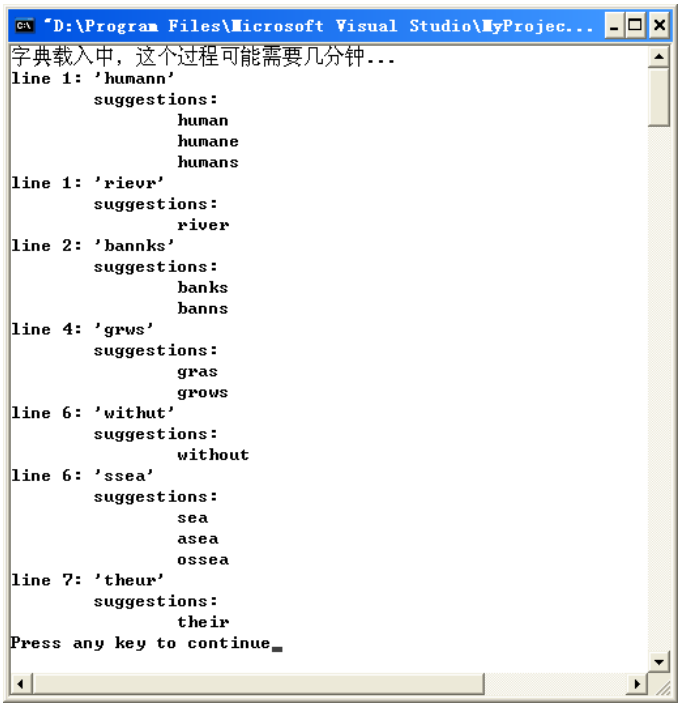


图 10-26　程序运行结果

读者可以在本书的在线支持资源中下载得到主程序文件的源代码清单。限于篇幅，此处不能详细列出，但为便于读者理解，我们稍做解释。主程序是整个程序的驱动所在。

main()函数中首先读入待检查文件“ test.txt ”，然后读入字典。其中通过一个含参数的构造函数来构造 Dictionary 对象。然后调用拼写检查函数 check_spelling()，该函数的实现中除了对文本文件中的拼写错误进行检查以外，同时还调用了 4 个提供修正意见的函数。这 4 个函数与前面所讲过的 4 条处理方式相对应。

下面给出 dictionary.h 文件的源代码清单。请读者注意该文件中提供的散列函数，该函数使用了一种非常简单的处理规则，应该不会给读者造成什么理解上的困难。此外，字典类 Dictionary 中最重要的就是该类的构造函数。它通过读取一个以文本形式提供的词汇表来初始化字典对象。请读者注意该类的实现方法。

```
#ifndef  _DICTIONARY_H_
#define  _DICTIONARY_H_

//此处略去了部分所需的头文件
#include  "hashset.h"
#include  "hashset.cpp"

using namespace std;

//散列函数
class hash_function
{
public:
   hash_function() {}

   unsigned int operator()( const string& s )  const {

         unsigned int res=0;
         for (int i=0; i<s.size(); i++)
             res=res*1000+s[i];
         return res;
   }
};

class equality
{
public:
   equality() {}
   bool  operator()( const string& A, const string& B )  const {
         return  (A == B);
   }
};

//字典类
class Dictionary: public HashSet<string, hash_function, equality> {

protected:
    ifstream infile;
public:
    Dictionary(const string &filename) {

         infile.open(filename.c_str());
         if ( !infile ) {
             cerr << "cannot open " << filename << " for input" << endl;
             return;
         }
```

```
        string line="";
        while (getline(infile,line)) {
            insert(line);
        }
        infile.close();
    }
};
#endif
```

读者应该已经注意到了 Dictionary 类继承了 HashSet 类。下面给出 HashSet 类的声明部分，它位于 hashset.h 文件中。

```
#ifndef  _HASHSET_H_
#define  _HASHSET_H_

//此处略去了部分所需的头文件
using namespace std;

template<class key_type, class hash_func, class key_equal>
class  HashSet {

protected:

   class Entry {
      public:
      key_type key;
      bool used;

      Entry() : used(false) {}
   };

   int entries;
   int prime;

   vector<Entry> *ht;
   hash_func hf;
   key_equal eq;

   int table_size()  const {return prime_list[prime];}
   float load_factor() const {return float(size()) / table_size();}
   int resize();

public:

   HashSet() : entries(0), prime(0),
        ht(new vector<Entry>(prime_list[0])){}

   virtual ~HashSet() {
        delete ht;
    }

   virtual int size() const {return entries;}
   virtual bool search(const key_type& k);
   virtual void insert(const key_type& k);
   virtual void remove(const key_type& k);
};

#endif
```

最后给出 hashset.cpp 文件的源码清单。该文件中包含了 HashSet 类的实现代码。可以看到 HashSet 类的实现中使用了模板。

```
#include  "hashset.h"

using namespace std;

static const int num_primes = 25;
static const unsigned long prime_list[] = {
   53, 97, 193, 389, 769, 1543, 3079, 6151, 12289, 24593, 49157, 98317,
   196613, 393241, 786433, 1572869, 3145739, 6291469, 12582917, 25165843,
   50331653, 100663319, 201326611, 402653189, 805306457
};

template<class key_type, class hash_func, class key_equal>
bool HashSet<key_type,hash_func,key_equal>::search(const key_type& k) {

   int p = hf(k) % table_size();

   while ((*ht)[p].used) {
      if (eq((*ht)[p].key, k)) return true;
      p++;
      if (p == table_size()) p = 0;
   }
   return false;
}

template<class key_type, class hash_func, class key_equal>
void HashSet<key_type,hash_func,key_equal>::remove(const key_type& k) {

   int p = hf(k) % table_size();

   while ((*ht)[p].used) {
      if (eq((*ht)[p].key, k)) {
         (*ht)[p].used = false;
            entries--;
            break;
      }
      p++;
      if (p == table_size()) p = 0;
   }
}

template<class key_type, class hash_func, class key_equal>
void HashSet<key_type,hash_func,key_equal>::insert(const key_type& k) {

   if (load_factor() > .7) resize();

   int pp = hf(k) % table_size();
   int p = pp;

   while (p < table_size() && (*ht)[p].used) {
      p++;
   }

   if (p == table_size()) p = 0;
   while ((*ht)[p].used) {
      p++;
   }
```

```
    (*ht)[p].key = k;
    (*ht)[p].used = true;
    entries++;
}

template<class key_type, class hash_func, class key_equal>
int HashSet<key_type,hash_func,key_equal>::resize() {

    if (prime == num_primes - 1) {
        cerr << "maximal table size reached, aborting ... " << endl;
        exit(2);
    }

    int mm = prime_list[prime];
    prime++;
    int m = prime_list[prime];
    vector<Entry>* ptr = new vector<Entry>(m);

    for (int i = 0; i < mm; ++i) {

        if ((*ht)[i].used) {
            key_type kk = (*ht)[i].key;
            int p = hf(kk) % m;

            while (p < m && (*ptr)[p].used) {
                p++;
            }
            if (p == m) p = 0;
            while ((*ptr)[p].used) {
                p++;
            }

            (*ptr)[p].key = kk;
            (*ptr)[p].used = true;
        }
    }

    delete ht;
    ht = ptr;
    return  m;
}
```

完成编码后，编译并运行程序即可。

10.6　不相交集

不相交集（Disjoint Set）又称并查集，是解决等价问题的一种有效的数据结构。树和森林常常被用来描述不相交集。本章前面已经研究了集合的基本理论及其应用。本部分内容将在此基础上做更为深入的探讨。

10.6.1　不相交集的概念

相信读者对“ 关系 ”这个词并不陌生，关系可以说是无处不在的。数学上，两个数值之间有“ 相等关系 ”，三角形之间有“ 相似关系 ”；生活中我们也常常遇到“ 同事关系 ”、

“合作关系”和“亲戚关系”等；在企业管理中，还会遇到部门之间的“隶属关系”和上下级之间的“管理与被管理关系”。易见，这些关系具有两个基本特征：一方面，涉及两个个体，即以两个个体x和y形成的客体(x,y)为讨论对象；另一方面，讨论对象(x,y)所满足的某种关系，应该是有明确含义的。

在集合论中，关系同样是一个非常重要的话题。通常把$A\times B=\{(a,b)|a\in A,b\in B\}$的一个子集$R$称为从$A$到$B$的关系。若$(a,b)\in R$，则称$a$与$b$有关系R，记为$a\mathrm{R}b$。等价关系是集合论中研究的最重要的一种关系。所谓等价关系（通常用符号“~”来表述）就是指同时满足下列 3 个性质的关系 R。

- 自反性：对于所有的$a\in S$，$a\mathrm{R}a$。
- 对称性：$a\mathrm{R}b\Leftrightarrow b\mathrm{R}a$。
- 传递性：若$a\mathrm{R}b$且$b\mathrm{R}c$，则有$a\mathrm{R}c$。

给定一个等价关系，一个显而易见的问题就是对于任意的元素a和b，能否确定$a\sim b$。例如，设在5个元素的集合$\{a_1,a_2,a_3,a_4,a_5\}$上定义一个等价关系。此时存在25对元素，其中每一对元素之间或者有关系或者没关系。但事实上，我们显然并不需要对全部这25个元素对之间的关系一一讨论。如果可以确定$a_1\sim a_2$，$a_2\sim a_4$，$a_3\sim a_4$，$a_5\sim a_1$便可根据等价关系的定义旋即推知全部元素对之间的关系。

再比如假设某区域内有一群人，人与人之间的关系要么是朋友，要么是敌人。而且朋友的朋友还是朋友，敌人的朋友仍是敌人。依据这样简单的规则，即使没有一张用于储存其中任意两人之间关系的庞大矩阵或者图表，理论上我们仍然可以轻松确定其中任意两人之间到底是朋友关系还是敌人关系。

元素$a\in S$的等价类是S的子集，它包含所有与a有等价关系的元素。回忆第 9 章中曾经介绍过的概念，易知等价类形成了对S的一个划分。S中的每一个元素都恰好出现在一个等价类中。为确定$a\sim b$是否成立，只需要验证a和b是否在同一等价类中。

如果说两个集合是互不相交的，其实就是指这两个集合不存在公共的元素。假设现在有N个集合，其中任意两个集合都是不相交的，即$S_i\cap S_j=\emptyset$，其中$i\neq j$，则这N个集合就构成了一个不相交集。可见，不相交集中的元素仍然是集合。初始状态下，作为不相交集元素的各个集合中仅包含有一个元素，如图 10-27 所示。

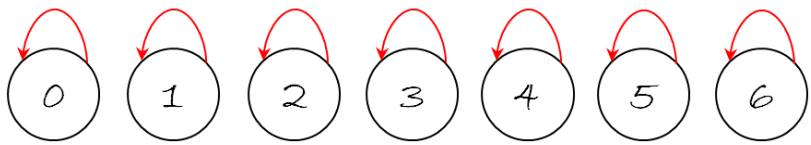


图 10-27　初始不相交集

不相交集上可以定义两种基本的操作。第 1 种是查找（Find），它返回包含给定元素的集合（即等价类）的名字。第 2 种操作是添加关系。如果想要在元素a和b之间建立一个等价关系，那么就要看这两个元素是否已经在一个等价类中，这一点可以通过对这

两个元素执行查找操作来实现。如果它们并不在同一个等价类中，那么就使用并（Union）操作。这种操作将把含有a和b的两个等价类合并成一个新的等价类。从集合的角度来说，并操作的结果是建立一个新的集合$S_k = S_i \cup S_j$，同时删除原来的两个集合从而保持所有的集合仍然彼此互不相交。正是因为不相交集上定义了这两种基本操作——并和查找，所以不相交集才又被称为并查集。

树和森林常常被用来描述不相交集。可以使用单独的一棵树来描述一个集合，因为树中的每一个元素都有相同的根。这样便可使用根来标识不同的集合。而且对处于同一集合中的两个元素执行 Find 操作，我们所期待的返回值应该是相同的名字。显然，树形结构的特点可以很好地符合这种需求。如果用单独的一棵树来表示并查集中的一个集合，那么树的集合就会形成森林。

初始状态下，每个集合中都只含有一个元素，如图 10-27 所示。集合的名字由树中的根节点给出。此时，节点中对我们有意义的信息除了节点本身所包含的值以外，就是一个指向其父亲节点的指针。由此，我们从任何一个节点出发，都可以逐层向上追溯，最终找到该树的根节点。在图 10-27 中，由于每个节点同时又都是根节点，所以其中指向父亲节点的指针都指向了其自身。

为了执行两个集合的 Union 操作，可以通过将一棵树的根链接到另外一棵树中的某个节点下（例如根节点），即成为另外一棵树中该节点的孩子节点。显然，这种操作花费常数时间。如图 10-28 所示，若想将节点 3 和节点 4 所在的集合进行合并，则可将节点 4 所在的树的根节点（就是其自身）中指向父亲节点的指针链接到节点 3 上。

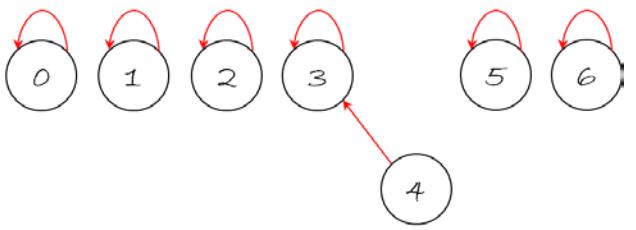


图 10-28　不相交集的并操作 1

如图 10-29 所示为将节点 5 和节点 6 所在的集合进行合并后得出的结果。

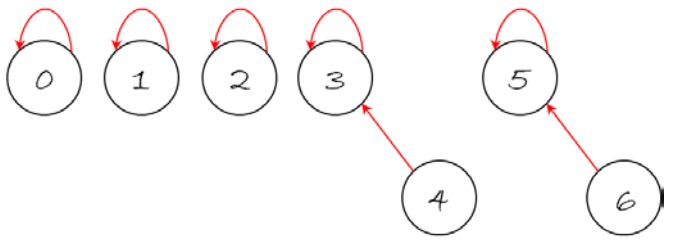


图 10-29　不相交集的并操作 2

如图 10-30 所示为将节点 6 和节点 4 所在的集合进行合并后得出的结果。此处，节点 6 所在的树的根节点是 5，于是我们将节点 5 中指向父亲节点的指针链接到节点 3 上。

这里需要强调，在将两个节点所在的集合进行合并时，至少是将其中一棵树的根节点链接到另外一棵树中的某一节点上，而这个节点也不一定非得是根节点。如果仅从功能实现的角度来看（不考虑执行的效率），将其中一棵树的根节点链接到另外一棵树中时，无论链接到对方的哪个节点显然都可以完成合并集合的任务。

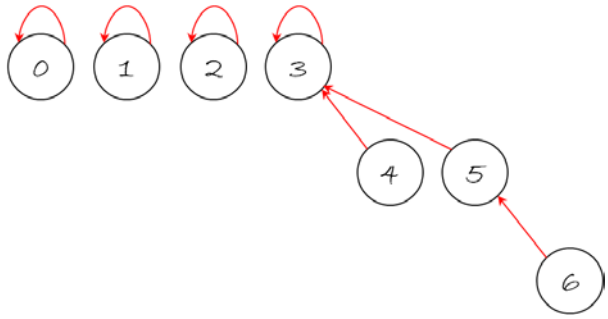


图 10-30　不相交集的并操作 3

对元素x执行一次 Find(x)操作通过返回包含x的树的根来完成。执行这种操作的时间消耗与x节点的深度成正比，当然前提是找到x节点本身的时间代价是一个常数。基于这一观点，在实现不相交集算法时，就有一些可以优化的策略。

10.6.2　不相交集的实现

根据 10.6.1 节中的介绍，下面就来实现不相交集类，代码如下。稍后我们将以此为基础展开讨论。

```
class Disjoint_set{
    unordered_map<int, int> PARENT;
public:
    Disjoint_set(int * array, int len){
        for(int i = 0; i < len; i++)
            PARENT[array[i]]=array[i];
    }

    int Find(int item){
        if(PARENT[item] == item)
            return item;
        else
            return Find(PARENT[item]);
    }

    void Union(int node1, int node2){
        int set1 = Find(node1);
        int set2 = Find(node2);
        PARENT[set1]=set2;
    }
};
```

上述代码中并操作的执行是比较低效的。它通过使第 1 棵树成为第 2 棵树的子树而完成合并，也就是让第 1 个节点所在树的根节点成为第 2 个节点所在树的根节点的孩子。若想对其加以改进，最先想到的方向就应该是设法打破这种随意性。例如，总是令较

小的树成为较大的树的子树（树的大小即其中所含节点个数的多少），这种方法也被称为是按大小求并。

在图 10-30 的基础上，现在要执行 Union(3, 2)操作，如果是按照上述代码中的策略，则会得到如图 10-31 所示的结果。显然如此下去，将会得到一棵越来越深的树，而前面我们已经讨论过，Find(x)函数的执行时间消耗与节点 x 的深度成正比。所以一个越来越深的树显然不是我们所期待的。

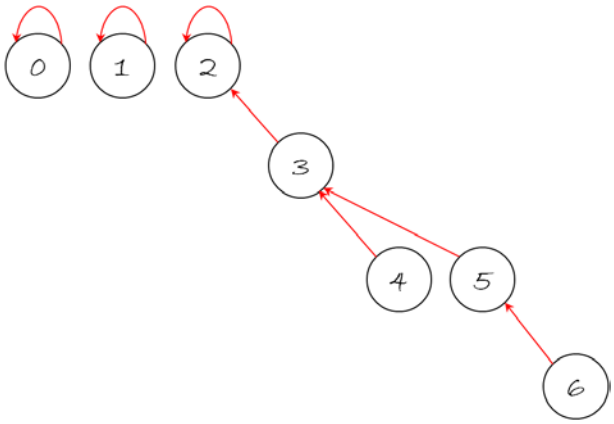


图 10-31　不加优化的并操作

但是如果预先对树的大小进行探查，再按大小求并，那么执行的结果就完全不同了。如图 10-32 所示，令较小的树成为较大的树的子树，无疑可以更好地控制树深的增长。更进一步还可以证明，如果并操作都是按大小进行的，那么任何节点的深度均不会超过$\log N$。这也意味着，Find 操作的运行时间是$O(\log N)$，连续M次操作的时间消耗即为$O(M \log N)$。

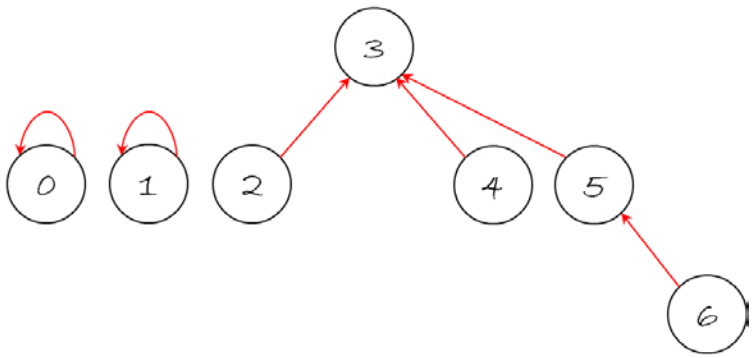


图 10-32　按大小执行并操作

为了实现这种方法，就需要记录每一棵树的大小。具体实现时可采用的方案很多，而且这与我们采用何种结构描述不相交集有关。例如，按照本节开始所给出的实现代码，再增加一个字典用以存储每棵树的大小显然是可行的。但是否还有更好的选择呢？在前面的代码中，我们使用一个〈键值-属性〉对来记录每一个节点。特别地，根节点的

属性项记录的内容是其自身，而其他节点的属性项记录的内容是它的父亲节点。不妨利用根节点的属性项来标记树的大小。同时为了区分根节点与其他节点，可以令这个标记树大小的值为负数。起初，每个树中仅含有一个节点，所以这时该属性项值为-1。

另一种实现方法为按高度求并，它同样保证所有的树的深度最多是$O(\log N)$。在这种方案中，程序记录每棵树的高度（而非大小），在执行并操作时令更浅的树成为更深的树的子树。如此一来，只有当两棵相等深度的树求并时，结果树的高度才会增加，而且树高的增加为1。在编程实现时，由于用负数来标记树深，所以代码中的处理方式是减1。按高度求并是按大小求并的一种轻量级修改。下面给出基于按深度求并改进的不相交集类，按大小求并的实现代码与此类似。

```
class Disjoint_set{
    unordered_map<int, int> PARENT;
public:
    Disjoint_set(int * array, int len){
        for(int i = 0; i < len; i++)
            PARENT[array[i]] = -1;
    }

    int Find(int item){
        if(PARENT[item] < 0)
            return item;
        else
            return Find(PARENT[item]);
    }

    void Union(int node1, int node2){
        int set1 = Find(node1);
        int set2 = Find(node2);
        if(set1==set2)return;
        if(PARENT[set1] < PARENT[set2])
            PARENT[set2] = set1;
        else
        {
            if( PARENT[set1] == PARENT[set2])
                PARENT[set2]-=1;
            PARENT[set1]=set2;
        }
    }
};
```

本书的在线支持网站上另提供有上述不相交集类的测试程序，读者可以自行下载后用于验证上述函数的使用。在图 10-30 的基础上，执行 Union(3, 2)操作，无论是采用按大小求并还是按高度求并的方法都会得到如图 10-32 所示的森林。图 10-33 中的(a)图是按高度求并后并查集森林的隐式表达。图 10-33 中的(b)图是按大小求并后并查集森林的隐式表达。

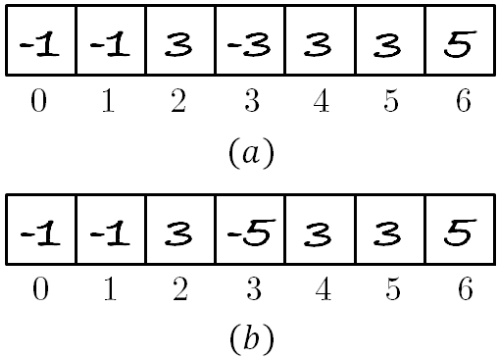


图 10-33　并查集森林的隐式表达

10.6.3　犯罪团伙的问题

经典问题求解

犯罪团伙问题，请你协助警方对某地的犯罪团伙组织情况展开分析。

不相交集在现实中有很多应用。本节将利用之前给出的不相交集类来解决犯罪团伙问题。现在警察正在对一个区域内的犯罪团伙展开调查，并搜集到了一些线索。已知有 10 名嫌疑犯，并分别用 0～9 标记：其中 0 号与 1 号是同伙；2 号与 3 号是同伙；4 号与 1 号是同伙；3 号与 5 号是同伙；1 号与 5 号是同伙；7 号与 6 号是同伙；8 号与 6 号是同伙；0 号与 5 号是同伙；1 号与 3 号是同伙。罪犯同伙的同伙也是同伙，请问该区域一共有多少个独立的犯罪团伙？

这显然是一个需要用并查集进行求解的问题，下面给出程序源码供读者参考。

```
#include <iostream>
#include <unordered_map>

using namespace std;

int main(int argc, char** argv) {

    int array[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    int len = sizeof(array)/sizeof(array[0]);
    Disjoint_set dj_set(array, len);

    dj_set.Union(0, 1);
    dj_set.Union(2, 3);
    dj_set.Union(4, 1);
    dj_set.Union(3, 5);
    dj_set.Union(1, 5);
    dj_set.Union(7, 6);
    dj_set.Union(8, 6);
    dj_set.Union(0, 5);
    dj_set.Union(1, 3);

    int nums[] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
    for(int i = 0; i < len; i++)
        nums[dj_set.Find(i)]+=1;

    int sum = 0;
    for(int i = 0; i < len; i++)
    {
        if(nums[i] != 0)
            sum++;
```

```
    }
    cout<<"犯罪团伙数量："<<sum<<endl;

    return 0;
}
```

输入代码后，编译并执行程序即可。

算法

并查集的路径压缩算法。

10.6.4 路径压缩的实现

到目前为止，我们所介绍的 Union/Find 算法对于大多数的情形都是完全适用的，它们不仅非常易于实现，而且对于连续个指令，其平均的时间消耗也是线性的。但是，最坏的情况还是难以避免而且很容易发生的。那么前面给出的算法所面对的最坏情况是什么呢？

我们已经讨论过，当两棵树大小（或者高度）相同的时候，并操作会导致结果树的高度加1。最糟糕的情况就是每次合并的两棵树大小（或者高度）都相同，例如如图 10-34 所示的情况就属于这种类型。不要忘了，连续M次 Find 操作的时间消耗为$O(M \log N)$。对于最坏的情况这样的结果显然不是我们所能接受的。

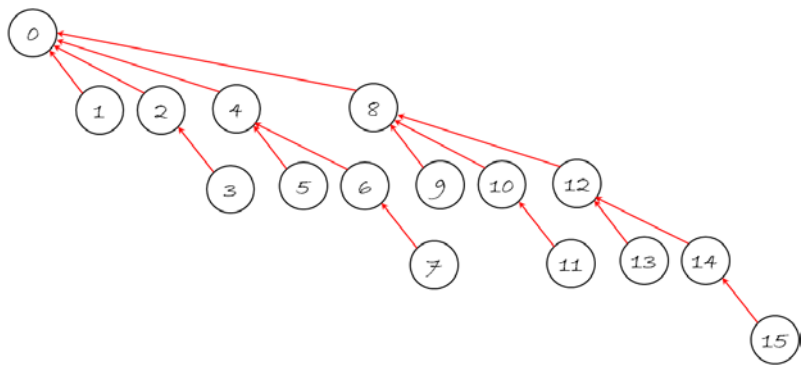


图 10-34 最坏的情况

对于 Union 算法可以改进的空间已经非常有限。执行 Union 操作的任何算法都将产生相同的最坏情形的树，因为它必然会任意打破树间的均衡。因此，无须对整个数据结构重新加工而使算法加速的唯一方法是对 Find 操作做些更巧妙的改进。这也就是本节要讨论的路径压缩算法。

路径压缩算法在一次 Find 操作期间执行，而与用来执行 Union 的方法无关。设操作为 Find(x)，此时路径压缩的效果是，从x节点到根的路径上的每一个节点都使它的父节点变成根。图 10-35 给出对图 10-34 中最坏情况下的树执行 Find(14)后路径压缩的效果。

路径压缩的效果在于通过额外的两个链的变化，节点12和节点 13离根近了一个位置，而节点14和节点 15现在离根近了两个位置。因此，以后对这些节点的访问将（由于当前的路径压缩处理）变得更加快捷。为了实现路径压缩算法，我们甚至无须对基本的 Find 操作进行大的改动。正如下面的代码所示的那样，唯一的变化是使得 PARENT[item]

等于由 Find 返回的值。这样，在集合的根被递归地找到以后，item 的父链就更新了原有值。由于通向根的路径上的每一个节点都会递归地出现，因此实现了路径压缩。

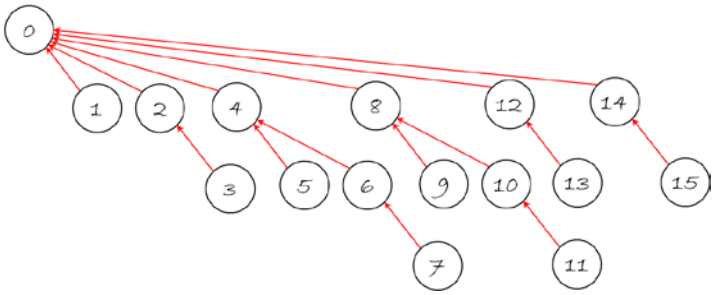


图 10-35　路径压缩算法示例

```
int Find(int item){
    if(PARENT[item] < 0)
        return item;
    else
        return PARENT[item] = Find(PARENT[item]);
}
```

任意执行 Union 操作的时候，路径压缩是一个好的方法。当有许多深层节点存在时，可通过路径压缩将它们移近根节点。研究人员已经证明，在这种情况下进行路径压缩时，连续M次操作最多需要$O(M \log N)$的时间。不过，在这种情况下确定平均情形的性能如何仍然是一个尚未解决的问题。

路径压缩不完全与按高度求并兼容，因为路径压缩可以改变树的高度。我们根本不清楚如何有效地重新计算它们。事实上，我们也不进行这样的计算。此时，对于每棵树所存储的高度是估计的高度（有时称为秩），但按秩求并理论上和按大小求并效率是一样的。不仅如此，高度的更新也不如大小的更新频繁。与按大小求并一样，我们也不清楚路径压缩平均来讲是否值得。但可以明确的是，使用求并探查法或路径压缩都能够显著地减少最坏情形运行时间。

10.7　STL 中的 set

在 STL 中容器 set 用以实现集合的功能。容器 set 与容器 vector 有很多相似之处，但是容器 set 中不能存储重复的元素，这是它与 vector 之间的区别。此外，set 中的元素是有序排列的，这一点与 map 相同。

可用如下语句声明一个集合对象。

```
set<int> mySet;
```

函数 insert()用于向 set 对象中插入新元素。例如，可利用下面的语句向集合 mySet 中添加整数 1 与 2。

```
mySet.insert(1);
mySet.insert(2);
```

此外，容器 set 中还提供了 size()和 find()等成员函数。下面这段示例代码演示了这些常用函数的使用方法。注意在使用 set 之前必须包含相应的头文件。

```
#include<set>
#include<iostream>
#include<algorithm>

using namespace std;

void main()
{
    set<int> mySet2;
    set<int> mySet3;

    set<int>::iterator i;

    int a[] = {1, 2, 3, 4, 5};

    set<int> mySet1( a, a + 5);

    cout<<mySet1.size()<<endl;

    i = mySet1.begin();
   cout<<"mySet1 : ";
    //遍历输出
   while( i != mySet1.end() )
   {
        cout<< *i <<" ";
        ++ i;
   }

    mySet2.insert(2);
    mySet2.insert(3);
    mySet2.insert(5);
    mySet2.insert(7);

    cout<<endl;

    if (mySet2.find(5) != mySet2.end()) {
        cout << "mySet2 contains 5";
    }

    ostream_iterator<int> out(cout, " ");
    //求集合 mySet1 与 mySet2 的交集
    cout << "\nmySet1 与 mySet2 的交集: ";
    set_intersection(mySet1.begin(), mySet1.end(),
                              mySet2.begin(), mySet2.end(),out);
    //求集合 mySet1 与 mySet2 的并集
    cout << "\nmySet1 与 mySet2 的并集: ";
    set_union(mySet1.begin(), mySet1.end(),
                              mySet2.begin(), mySet2.end(), out);
    //求集合 mySet1 与 mySet2 的差
    cout << "\nmySet1 与 mySet2 的差集: ";
    set_difference(mySet1.begin(), mySet1.end(),
                              mySet2.begin(), mySet2.end(),out);
    //求集合 mySet1 与 mySet2 的对称差
    set_symmetric_difference(mySet1.begin(), mySet1.end(),
                              mySet2.begin(), mySet2.end(),
```

```
                                inserter(mySet3, mySet3.begin()));

    cout << "\nmySet1与mySet2的对称差: ";
    i = mySet3.begin();
    while( i != mySet3.end() )
    {
        cout<< *i <<" ";
        ++ i;
    }

    cout<<endl;
}
```

上述代码中，对象 mySet2 和 mySet3 使用无参数构造函数生成。对象 mySet1 使用含参数构造函数生成。其参数通过数组中元素的地址来截取该数组元素，并以此来初始化 set 类对象，注意该构造函数的参数必须是两个，参数 1 是待截取数组的起始地址，参数 2 是待截取数组的结束地址。

函数 size()用于计算集合的大小。

函数 find(const Key & key)用于在集合中搜索特定元素。若目标元素存在于集合中，则该函数返回一个迭代器，指向被控序列中键值与 key 相等的元素。若不存在这样的元素，函数将放回 end()，end()函数返回的是标志 set 容器结束的迭代器。同理，成员函数 begin()能够返回容器中的首元素的迭代器。

遍历存储在 set 对象中的元素可以使用迭代器 iterator。具体方法读者可以参考上述代码中的相关部分。

函数 set_intersection、set_union、set_difference 及 set_symmetric_difference 分别用来实现集合间的交、并、差及对称差。请读者注意上述函数的定义并不位于头文件<set>中，而是位于头文件<algorithm>中，因此在使用之前必须包含该头文件。

对称差是指这样一个集合，该集合中的元素仅属于两个输入序列中的一个，而不包括两个输入序列交集中的元素。若集合A为$\{1,2,3\}$，集合B为$\{2,3,4\}$，则A与B的对称差为集合$\{1,4\}$。

示例代码中使用上述函数时采用了两种不同的方式。第 1 种使用的方式如下。

```
set_intersection(mySet1.begin(), mySet1.end(),
                      mySet2.begin(), mySet2.end(), out);
```

其中 out 是输出迭代器，用于记录计算结果。

第 2 种使用的方式如下。

```
set_symmetric_difference(mySet1.begin(), mySet1.end(),
                                mySet2.begin(), mySet2.end(),
                                    inserter(mySet3, mySet3.begin()));
```

这个方法把运算结果赋给了另外一个 set 对象。

请读者注意，上述 4 个函数中的任意一个都具有这两种使用形式。此外，以上 4 个函数并不专属于 set 类，它们是 STL 算法（algorithm）的组成部分。对类 vector 与类 queue

同样可以应用以上函数。但这 4 个函数与 set 类同用效率会更高。

本章参考文献

[1] 耿素云，屈婉玲，张立昂．离散数学（第 3 版）．北京：清华大学出版社，2004.

[2] 刘立康，黄力宇，胡力山．微机原理与接口技术．北京：电子工业出版社，2010.

[3] Brian W. Kernighan，Dennis M. Ritchie．徐宝文，李志，译．C 程序设计语言（第 2 版）．北京：机械工业出版社，2004.

[4] Robert Sedgewick．霍红卫，译．算法：C 语言实现（第 1~4 部分）．北京：机械工业出版社，2009.

[5] Mark Allen Weiss．张怀勇，等译．数据结构与算法分析 C++描述（第 3 版）．北京：人民邮电出版社，2007.

[6] Bernard A. Galler, Michael J. Fischer. An Improved Equivalence Algorithm. Communications of the ACM, Vol. 7, No. 5: 301~303, 1964.

第 11 章

排序——有序让世界更美好

排序是计算机科学中的一种重要操作。有序的数据处于相对稳定的状态，更容易被计算机处理，因此对数据进行排序是非常有必要的。例如，在将数据写入报表之前，可能要对其进行排序操作；在大型数据库应用中，排序也必然会被用到；此外，有些数据查找算法要求待处理的数据呈升序（或降序）排列，彼时排序操作同样必不可少。简单来讲，排序是将一组数据按一定规律重新排列的过程。而排序算法就是针对不同数据的特点，所采取的特定的数据排列方法。

11.1 排序问题概述

小提示

实现某个具体的排序算法是笔试、面试中最常见的一类问题，所以如果你在准备应聘 IT 公司，那么这部分就应该成为你的重点。

大量统计调查显示，计算机在数据处理过程中，有 25%的时间花费在排序上。在很多计算机设备上，排序用去计算机处理数据时间的一半以上，而且排序不仅用于数据排序，还广泛用于网络路由选择、数字信号处理和数据库应用等诸多领域。

作为计算机程序设计中的一种重要操作，排序是信息处理、数据处理领域中经常使用的一种重要运算。它的功能是将一个数据元素的任意序列，重新排列成一个按关键字排列的有序序列。有效的排序方法是计算机应用中重要的研究课题之一。在具体讲述各种排序算法之前，这里首先给出一些预备知识。

11.1.1 基本概念和定义

1．数据表

数据对象是对客观事物的抽象表示，它是多数据复合组成的有意义信息载体。俗话说“物以类聚，人以群分”，具有相同性质聚集在一起的数据对象，更加便于批量处理，这就构成了所谓的数据集。这里将数据表定义为待排序的有限数据集。

2．关键码

通常一个数据对象不只有一个属性（数据项），关键码指的是数据对象中能起标识作用的数据项。比如图书馆的书目管理需要对图书依据其内容进行分类编号，尽管每本书都有书名。但书名有可能相同，而每本书的编号却是唯一的。那么书目表所依据的关键码就应该是图书编号，而书名可以作为次关键码。

通常把在数据表中各个对象之间互不相同的关键码称为主关键码。利用一种排序算法

按照数据表的关键码进行排序，排序的结果一般应该是唯一的。实际上，主关键码的功能相当于数据库系统中的表的主键，主键能唯一地标识一条记录，主关键码则能唯一地标识一个数据库对象。但有可能出现关键码相同的情况，那么就把这种关键码称为次关键码。按照次关键码利用一种排序算法进行排序，结果可能不唯一。例如，对班级学生成绩表按成绩进行排名，不同学生的成绩有可能是相同的，排序时这些学生在一起，谁在前谁在后都有可能。所以在不同情况下，关键码的选取是很重要的。

3．排序的数学定义

设含有n个对象的序列为$\{R[0],R[1],\cdots,R[n-1]\}$，该序列所对应的关键码序列为$\{K[0],K[1],\cdots,K[n-1]\}$。所谓的排序，就是确定$0,1,\cdots,n-1$的一种排列$p[0],p[1],\cdots,p[n-1]$使各关键码满足如下的非递减（或非递增）关系：$K[p(0)]\leqslant K[p(1)]\leqslant\cdots\leqslant K[p(n-1)]$或$K[p(0)]\geqslant K[p(1)]\geqslant\ldots\geqslant K[p(n-1)]]$。也就是说，所谓排序，就是按照一定的算法把数据对象依次排列起来，使一组按关键码任意排列的对象变成按关键码线性有序排列的对象。

11.1.2 排序算法的分类

1．稳定排序和非稳定排序

在待排序的记录中，如果存在多个关键码相同的记录，经排序后这些记录的相对次序仍然保持不变，即排序后这两个对象具有与排序前相同的顺序，那么就称相应的排序方法是稳定的，否则就是不稳定的。需要注意的是不能简单地认为不稳定的排序方法没有稳定的排序方法好，这是没有依据的。

2．内排序与外排序

由于待排序的记录数量不同，使得排序过程中涉及的存储器不同，因而可以根据在排序过程中数据对象是否完全存放在内存中，而将排序算法分为两大类：一类称为内部排序（Internal Sort），指的是待排序记录存放在计算机随机存储器中进行的排序过程；另一类是外部排序（External Sort），指的是在排序期间因为数据对象过多或过大，以致内存一次不能容纳全部记录，因而在排序过程中需要对外存进行访问的排序过程，也就是会引起数据对象在内外存之间移动的排序。本章所述的排序算法仅限于内部排序。

内部排序的方法很多，就其全面性能而言，很难提出一种被认为是最好的方法。每一种方法都有各自的优缺点，适合在不同的环境下使用。对于不同排序算法的分类方案不同文献上的说法非常不统一，因此读者大可不必在此问题上细究。本书按照排序所用策略的不同对内部排序方法进行分类，大致分为如下 5 类：（1）插入排序法；（2）交换排序法；（3）选择排序法；（4）归并排序法；（5）分配排序法。

由于排序算法众多，本章限于篇幅不能一一尽述，本章后续部分将选择 10 种非常具有代表性的算法进行介绍，它们是直接插入排序、二分插入排序、希尔排序、直接选择排序、堆排序、冒泡法排序、鸡尾酒排序、快速排序、归并排序和计数排序。

11.1.3　排序算法的分析

求职面经点拨

本书附录题库中的第31题至第33题是一些与排序有关的面试题，有兴趣的读者可以在完成本部分内容学习后，试回答这些问题。

大多数排序算法都有两个基本的操作：比较两个关键字的大小和改变指向记录的指针或移动记录本身。而第 2 种操作的实现依赖于待排序记录的存储方式。这里简单说明一下待排数据的常用存储方式，主要有：

- 以顺序表作为存储结构，此时排序的过程是对记录本身进行物理重排（即通过关键字之间的比较判定，将记录移到合适的位置）。
- 以链表作为存储结构，此时排序的过程为无须移动记录，而仅需修改指针。通常将这类排序称为链表（或链式）排序。
- 用顺序的方式存储待排序的记录，但同时建立一个辅助表（如包括关键字和指向记录位置的指针组成的索引表）。此时排序的过程只需对辅助表的表目进行物理重排（即只移动辅助表的表目，而不移动记录本身）。适用于难于在链表上实现，仍需避免排序过程中移动记录的排序方法。

目前，基于上述两种基本操作构建的排序算法有几十种，如何衡量和分析这些算法的效能在实际应用中非常重要。通常排序的时间开销是衡量排序算法好坏的最重要标志。排序的时间开销可以用算法执行中的数据比较次数与数据移动次数来衡量。由于排序算法的执行效率与待排序集合的自身特点有关，通常，估算算法运行时间代价可以按平均情况来进行。特别地，对于那些受对象关键码初始状态和对象个数影响较大的情况，需要按最好情况和最坏情况进行估算。

除此之外，算法执行时所需的附加存储空间也是一个考察排序算法性能的指标。综合运用上述两点能够有效得到排序算法性能的一些定性评价。

排序算法如此之多，在实际应用中，如何选择适合的排序算法呢？完全依据O表达式的定量关系来选择显然欠妥。很多讲述算法的书籍常常对此问题避而不谈，因此很容易造成读者的误会和误解。实际上，影响排序效率的因素有很多，在多种条件影响之下，往往使得最优O表达式的算法不见得就是最快的。同样，最差O表达式的算法也不一定是最慢的。毕竟算法时间复杂性的渐近表示只是反映了当n增大时，程序执行时间的一种变化趋势，它并不能反映算法在实际工作时的效率。算法时间复杂性的渐近分析反映了算法本身和待排序数据量的关系。具有较好渐近特性的算法适合于合对大数据量的排序，但对小数据量的排序常常反而不及具有较差渐近特性的算法。此外，有的算法对排序数据原有的“有序性”很敏感，如插入排序和快速排序；有的算法对关键码的范围大小很敏感，如桶排序。因此选择排序算法时应当综合考虑多种因素的影响。

有文献总结了选择排序法的几个原则。首先，当数据量不大时选择插入或选择排序，不要用冒泡排序。其次，当数据量大而又注重空间复杂性时选择快速排序或堆排序。再次，当数据量大而又允许使用较多附加空间时选择桶排序。最后，当要在已排序数据上增加若干新数据时选择插入排序。

11.2 插入排序

插入排序是一类最简单、最朴素的排序方法。它的基本思想是：每次将一个待排序数据对象按关键码大小插入一个有序的数据序列中，得到一个新的容量加 1 的数据序列，如此往复直到全部对象插入完毕为止。不同的插入排序算法寻找插入位置的方法是不同的，这也是区别各种插入算法的关键所在。下面将介绍 3 种不同的插入排序方法。

11.2.1 直接插入排序

算法

十大排序算法之直接插入排序。

直接插入排序的基本思想是：数据对象在顺序表中存储，当插入第i个对象$V[i]$时，前面的$i-1$个元素$V[0],V[1],\cdots,V[i-1]$已经排好序，用第i个对象的关键码同已经存在的$i-1$个对象的关键码从后往前顺序进行比较，找到合适的位置以后就将$V[i]$插入，而插入点以后的元素都随之向后移动一位。

直接插入排序的过程可以进一步理解为：一个长度为i的数组被分为两个集合，即已排序集合和未排序集合。开始时已排序集合为空，而未排序集合即为整个数组。当排序开始后每插入一个对象，已排序集合元素数目加 1，相应地未排序集合的元素数目减 1，重复插入过程，直至将未排序集合清空，这时已排序集合就是最终结果。图 11-1 演示了数组的直接插入排序过程。

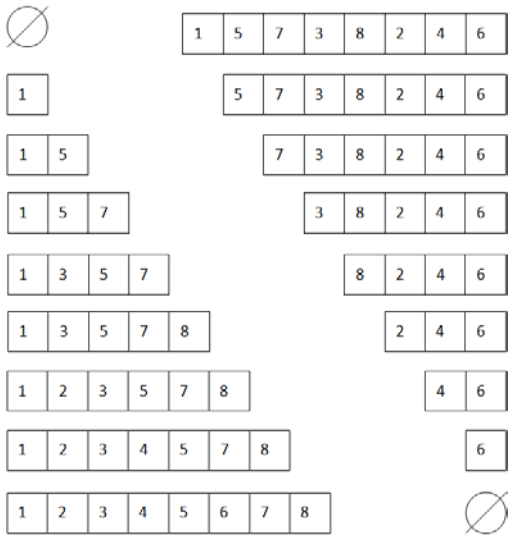


图 11-1 直接插入排序

在具体实现时，直接插入排序是由两层嵌套循环组成的。外层循环标识并决定待比较的数值。内层循环为待比较数值确定其最终位置。直接插入排序是将待比较的数值与它的前一个数值进行比较，所以外层循环是从第 2 个数值开始的。当前一数值比待比较数值大的情况下继续循环比较，直到找到比待比较数值小的并将待比较数值置入其后一位置，结束该次循环。

下面给出了实现直接插入排序算法的源代码。

```
#ifndef INSERTSORT_H
#define INSERTSORT_H

#include <vector>
using namespace std;

class InsertSort
{
private:
    int len;
    vector<int> list;
public:
    InsertSort(vector<int> _list, int _len);
    void insert_sort();
    void out();
};
#endif

#include "InsertSort.h"
#include <iostream>

using namespace std;

InsertSort::InsertSort(vector<int> _list, int _len)
{
    for (int i=0; i<_len; i++) list.push_back(_list[i]);
    this->len = _len;
}

//直接插入排序
void InsertSort::insert_sort()
{
    int insertNum;
    for (int i=1; i<len; i++)              //总共进行 len 次插入
    {
        insertNum = list[i];               //待插入元素
        int j = i;
        while (j>0 && insertNum<list[j-1]) //寻找插入位置
        {
            list[j] = list[j-1];
            j--;
        }
        list[j] = insertNum;
    }
}

void InsertSort::out()
{
    //输出排序后的数组，此处代码略
}
```

函数 Insert_sort()的外层 for 循环执行了len − 1次。内部的 while 循环最多执行j次，j可以从 1～$n-1$。在最好的情况下，排序前对象已经有序，每次只需与前面有序对象序列的最后一个关键码比较 1 次，移动 2 次对象。总的比较次数为$(n-1)$次，总的移动次数为$2(n-1)$次。在最坏情况下算法开销要大得多，处理第i趟时必须与前面i个对象进行关键码比较，并且 1 次比较就对应 1 次数据移动，算法的比较次数为：

$$1+2+\cdots+(n-1)=\frac{n(n-1)}{2}$$

算法的移动操作次数为：

$$1+\cdots+n-1+2(n-1)=\frac{(n+4)(n-1)}{2}$$

所以在最坏的情况下，直接插入排序算法的时间效率为$O(n^2)$。对于小数组，插入排序简单易行；对于大数组，插入排序效率过低，不宜采用。不难看出，直接插入排序是一种稳定的排序方法。

算法

十大排序算法之二分插入排序。以类似思想为出发点的还有一个二分搜索，读者也应该掌握。

11.2.2 二分插入排序

二分插入排序又叫折半插入排序，它同直接插入排序一样，是每次将一个数据对象插入已经排好序的顺序表中。设在顺序表中有对象序列$V[0],V[1],\cdots,V[n-1]$，其中$V[0],V[1],,\cdots V[i-1]$是已经排好序的对象。二分插入排序是用折半查找法寻找$V[i]$的插入位置，然后插入这个对象，同样需要$n-1$趟比较。折半查找法的使用是二分插入排序与直接插入排序的区别。

折半查找法较朴素的直接插入排序在效率上有了很大的增进。图 11-2 演示了如何使用折半查找法在目标数组中确定数字35的位置。首先，确定位于有序目标数组中间位置的数字为45。因为$45>35$，所以再将原数组折半，并取左半部分作为搜索目标。此时又因为$23<35$，于是选择子串的右半部分作为搜索目标。而$35<36$，即可确定35的位置应该在23与36之间。

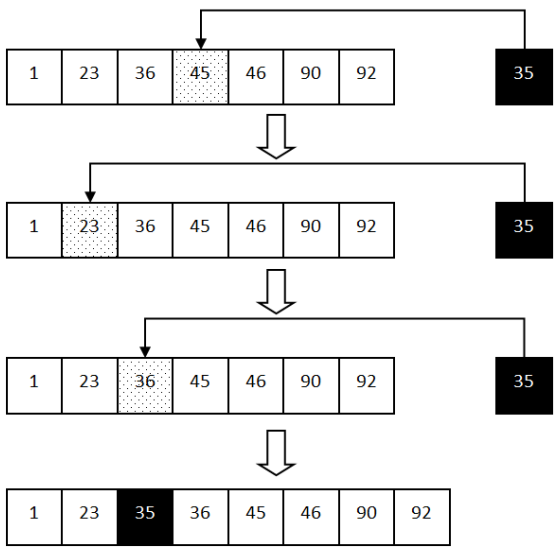


图 11-2 折半查找法

在本书第 6 章中曾经介绍过应用递归实现折半查找的方法，这里不再赘述。下面给出了实现二分插入排序算法的源代码。

```
// 二分插入排序
#ifndef BINARYINSERTSORT_H
#define BINARYINSERTSORT_H

#include <vector>
using namespace std;

class BinaryInsertSort
{
private:
    int len;
    vector<int> list;
public:
    BinaryInsertSort(vector<int> _list, int _len);
    void binary_insert_sort();
    void out();
};
#endif

#include "BinaryInsertSort.h"
#include <iostream>
using namespace std;
//构造一个顺序表
BinaryInsertSort::BinaryInsertSort(vector<int> _list, int _len)
{
    for (int i=0; i<_len; i++) list.push_back(_list[i]);
    this->len = _len;
}

void BinaryInsertSort::binary_insert_sort()
{
    int middle;
    for (int i=0; i<len; i++)
    {
        int insertNum = list[i];
        int left = 0;
        int right = i-1;
        while (left <= right)              // 二分法寻找插入位置
        {
            middle = (left + right)/2;
            if (insertNum > list[middle]) left = middle+1;
            else right = middle-1;
        }
        for (int j=i; j>left; j--) list[j] = list[j-1];
        list[left] = insertNum;
    }
}

void BinaryInsertSort::out() {
    //输出排序后的数组，此处代码略
}
```

二分查找比顺序查找快，所以二分插入排序就平均性能来说比直接插入排序要快。它所需要的关键码比较次数与对象个数紧密相关。插入第i个对象时，需要经过$\lfloor \log_2 i \rfloor + 1$次关键码比较，那么将$n$个对象，其中$n = 2^k$，用二分插入排序所进行的关键码比较次数为：

$$\sum_{i=1}^{n-1}(\lfloor \log_2 i \rfloor + 1) = \underbrace{1}_{2^0} + \underbrace{2+2}_{2^1} + \underbrace{3+\cdots+3}_{2^2} + \underbrace{4+\cdots+4}_{2^3} + \cdots + \underbrace{k+k+\cdots+k}_{2^{k-1}}$$

$$= (2^0 + 2^1 + \cdots + 2^{k-1}) + (2^1 + 2^2 + \cdots + 2^{k-1}) + (2^2 + 2^3 + \cdots + 2^{k-1}) + \cdots + 2^{k-1}$$

$$= \sum_{i=1}^{k}(2^k - 2^{i-1}) = k \cdot 2^k - \sum_{i=1}^{k} 2^{i-1} = n\log_2 n - n + 1 \approx n\log_2 n$$

可见当n较大时，二分查找算法的关键码比较次数比直接插入排序在完全逆序下的情况要好，比在完全有序下的情况要差。所以应根据不同情况灵活选用这两种方法。在对象的初始排列已经有序或者接近有序时，应用直接排序算法。至于二分插入排序和直接插入排序在对象移动次数上的区别，请读者自己思考。二分插入排序算法同直接插入排序一样，是稳定的排序方法。

11.2.3 希尔排序

算法

十大排序算法之希尔排序，以提出者名字命名的算法，克努特也曾在他的著作中对该算法进行过分析。

希尔排序由美国计算机科学家唐纳德 · 希尔（Donald L. Shell）于 1959 年提出，又称为缩小增量排序。该算法先取一个小于数据表中元素个数n的整数gap，并以此作为第 1 个间隔，将数据表分为gap个子序列，所有距离为gap的对象放在同一个子序列中。也就把数据表中的全部元素分成了gap个组。而所有距离为gap的倍数的记录会被放在同一个组中。分组确定后，就在每一个小组中分别进行直接插入排序。局部排序完成后就缩小间隔gap，并重复上述步骤，直至取到gap = 1时，完成最后一次直接插入排序。不妨简单地分析一下这个算法为什么会起作用。显然开始时间隔gap较大，因而各组中的数据量相对较小，而且至少在最开始的时候算法是非常快的。随着算法的进行，间隔gap取值变得越来越小，于是子序列中元素个数也就越来越多，所以排序工作可能会变慢。但是由于前面已经完成了部分排序工作，所以在很大程度上减轻了后期的工作量，最终总体的排序速度还是比较快的。这就是这种所谓“ 缩小增量排序 ”方法的设计原理所在。

还需要说明的是间隔gap的取法可能有很多种。后面的例子中所使用的取法是gap = ⌈gap/2⌉。最初希尔在设计算法时提出来的取法是gap = ⌊gap/2⌋。后来克努特提出取gap = ⌊gap/3⌋ + 1。也有学者认为取互质的gap为好。但到目前为止无论哪种取法都尚未得到严密的数学证明。

下面来举例说明希尔排序是如何工作的。假设有一个数据表{20,37,50,35,10,1}，现在考虑用希尔排序算法对其进行排序。显然，数据表中共有 6 个元素。那么第 1 轮操作可以取gap = ⌈gap/2⌉ = 3，于是将数据表中的元素分为 3 组，每组 2 个元素。分别对这 3 组进行直接插入排序，这个过程如图 11-3 所示。在第 2 轮操作时，首先将间隔缩小，取得gap = ⌈gap/2⌉ = 2，则将第 1 轮处理得到的数据表划分为间隔为 2 的 2 组，同理对其进行直接插入排序。这个过程如图 11-4 所示。接下来进行第 3 轮gap = 1时的操作。易见第 2 轮操作结束，数据表已经有序，因此事实上第 3 轮所做的工作就非常之少了，这里就不再赘述第 3 轮的操作过程了。

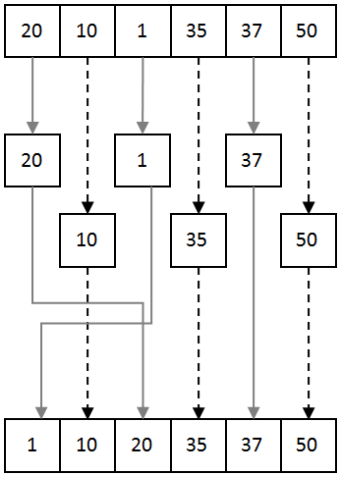


图 11-3　希尔排序示例第 1 轮操作

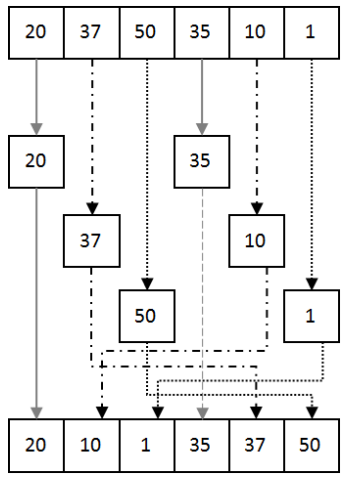


图 11-4　希尔排序示例第 2 轮操作

下面给出希尔排序算法的示例代码。

```
//希尔排序
#ifndef SHELLSORT_H
#define SHELLSORT_H

#include <vector>
using namespace std;

class ShellSort
{
private:
    int len;
    vector<int> list;
public:
    ShellSort(vector<int> _list, int _len);
    void shell_sort();
    void out();
};
#endif

#include "ShellSort.h"
#include <iostream>
using namespace std;

ShellSort::ShellSort(vector<int> _list, int _len)
{
    for (int i=0; i<_len; i++) list.push_back(_list[i]);
    this->len = _len;
}

void ShellSort::shell_sort()
{
    int insertNum;
    int gap = len/2;                          //初始增量
    while (gap)                               //当 gap>=1
    {
        for (int i=gap; i<len; i++)           //对 gap 间隔子序列进行插入排序
        {
```

```
            insertNum = list[i];            //待插入元素
            int j = i;
            while (j>=gap && insertNum<list[j-gap])
            {                               //寻找插入位置
                list[j] = list[j-gap];
                j -= gap;
            }
            list[j] = insertNum;            //插入
        }
        gap = gap/2;                        //缩小增量
    }
}

void ShellSort::out() {
    //输出排序后的数组，此处代码略
}
```

分析希尔排序的时间复杂度是一件十分困难的工作，特别在间隔选取方式发生变化的情况下，尚不能给出一个定量结果。到目前为止，关于间隔值选取方式与关键码比较次数和元素移动过程之间的关系不存在理论上的证明结果。最后需要说明的是希尔排序是一个不稳定的排序方法。

11.3 选择排序

选择排序方案是一类常用的排序方案。选择排序的基本思想是：在排序时每次选择最小项或最大项，将其放入适当位置，依此类推。好比玩扑克牌时要将扑克从小到大排列，最常用的方法就是在这套扑克牌中每次找出一张最大的牌，放入已选好的牌序列中，重复这个过程最终将完成扑克牌的有序排列。选择排序过程正是基于这样一种思想设计的。

11.3.1 直接选择排序

算法

十大排序算法之直接选择排序。

直接选择排序是最简单的选择排序方法，它的基本步骤是（以升序排列为例）：首先在一组数据对象$V[i],\cdots,V[n-1]$中选择具有最小关键码的对象，若它不是对象序列$V[i],\cdots,V[n-1]$中的第一个对象，则将它与这组对象中的第一个对象对调，然后在这组对象中剔除具有最小关键码的对象。在剩下的对象$V[i+1],\cdots,V[n-1]$中重复执行上述步骤，直到剩余对象只有一个为止。显然在这个过程中，具有相同关键码的对象可能会颠倒次序，因此直接选择排序算法是一种不稳定的排序方法。图 11-5 演示了算法的工作过程。

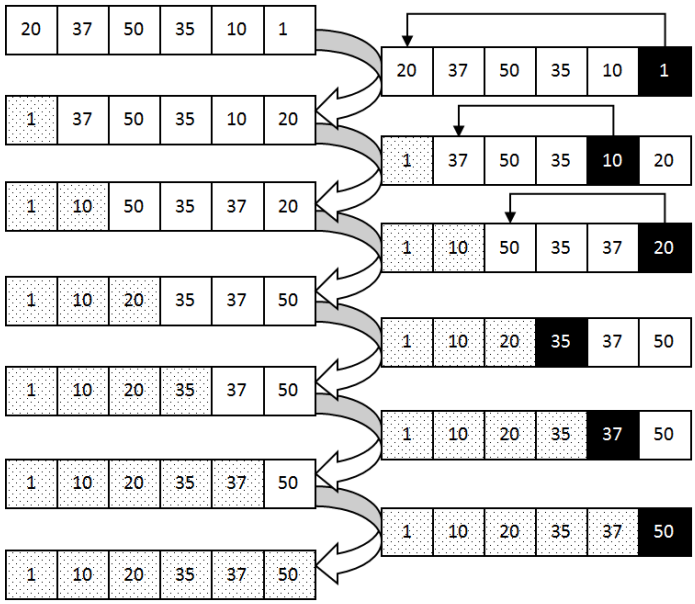


图 11-5　直接选择排序算法

下面给出实现直接选择排序算法的示例代码。

```
//直接选择排序
#ifndef SELECTSORT_H
#define SELCETSORT_H

#include <vector>
using namespace std;

class SelectSort
{
private:
    int len;
    vector<int> list;
public:
    SelectSort(vector<int> _list, int _len);
    void select_sort();
    void swap(int,int);
    void out();
};
#endif

#include "SelectSort.h"
#include <iostream>
using namespace std;

SelectSort::SelectSort(vector<int> _list, int _len)      // 构造一个数据集合
{
    for (int i=0; i<_len; i++) list.push_back(_list[i]);
    this->len = _len;
}

void SelectSort::select_sort()
{
    int k;
```

```
    for (int i=0; i<len; i++)                    //进行 len 次比较
    {
        k = i;
        for (int j=i+1; j<len; j++)              //找最小元素
            if (list[j] < list[k]) k = j;
        swap(i, k);                              //将最小元素放入 i 位置
    }
}

void SelectSort::swap(int i, int j)              //交换两元素位置
{
    int temp = list[i];
    list[i] = list[j];
    list[j] = temp;
}

void SelectSort::out() {
    //输出排序后的数组，此处代码略
}
```

由于直接选择排序要比较、交换和移动元素，这些操作往往要耗费比控制循环或操作数组更多的成本，所以在计算复杂度时忽略不计。函数 select_sort()中的外层 for 循环执行了$n-1$次，调用 swap()的次数是$n-1$，元素与临时变量之间的相互操作是$2(n-1)$次。而在第i次选择的时候，里层的 for 循环会执行$n-i-1$次，即进行$n-i-1$次比较。因此总的关键码比较次数是：

$$1+2+\cdots+(n-1)=\frac{n(n-1)}{2}$$

总的对象移动次数是$3(n-1)$次。所以可知n个项目的选择排序需要的主要操作次数为：

$$\frac{n^2}{2}+5\times\frac{n}{2}-3$$

因此可知选择排序的算法复杂度是$O(n^2)$。

选择排序不依赖于数据的初始排列，这是一个优势。但由于$O(n^2)$的增率很大，因此算法只适用于较小的n值。该算法需要$O(n^2)$次比较，而需要$O(n)$次数据移动。在数据移动成本高，比较操作成本低时，选择排序优于其他算法。如果每个数据项比较长，而排序关键字比较短，就可以考虑使用选择排序方法。

11.3.2 堆排序

算法

十大排序算法之堆排序，它是依托堆结构实现的一种排序算法。

堆这种数据结构在前面章节中已经介绍过，这里不再赘述。显然，如果将堆结构中的数据看作是一棵完全二叉树的各个节点，那么堆实质上就是满足如下性质的完全二叉树：树中任意非叶节点的关键字均不大于（或不小于）其左右孩子（若存在）节点的关键字。利用堆的有序性及其运算，可以很容易地实现选择排序。这种排序方法称为堆排序。

我们已经知道，堆可以分为两种，即最小堆和最大堆。其中最小堆又叫作小根堆，即根节点的关键字是堆里所有节点关键字中最小者的堆。而最大堆就是根节点的关键字

是堆里所有节点关键字中最大者的堆，最大堆又称为大根堆。为了实现数据表中元素的有序排列，需要建立最大堆。建立初始堆的过程这里不再赘述，读者可以翻阅本书前面的内容。

假设欲对含有n个对象的序列$V[0],V[1],\cdots,V[n-1]$进行堆排序，那么算法主要分为两个大步骤：一是根据初始输入数据利用堆的调整算法形成初始堆，二是通过一系列的对象交换和重新调整对堆进行排序。具体过程是在首次获得最大堆之后，将关键码最大的元素$V[0]$（即堆顶）和无序区的最后一个元素$V[n-1]$进行交换。由此得到新的无序区$V[0],V[1],\cdots,V[n-2]$和有序区$V[n-1]$，且满足$V[0,\cdots,n-2]$中元素的关键码不大于$V[n-1]$的关键码。由于交换后新的根$V[0]$可能违反堆性质，因此再次利用堆的调整算法对无序区$V[0],V[1],\cdots,V[n-2]$进行调整。调整完成后将关键码最大的元素$R[0]$和无序区的最后一个元素$V[n-2]$交换，并再次利用堆的调整算法对无序区进行调整……这个过程直到无序区中仅剩一个元素时为止。

图 11-6 演示了堆排序的工作过程。如图 11-6（a）所示的原始堆经过一次调整后，变成了最大堆。此时将关键码最大的元素与无序区中最后一个元素对调，即将图 11-6(b)中堆顶的50和1对调，结果如图 11-6（c）所示，此时50进入有序区。然后对无序区中的元素继续进行调整，使其仍然保持堆顶元素最大这一性质，如图 11-6（d）所示。重复之前的操作方法，直到排序完成，后续过程这里不再赘述。

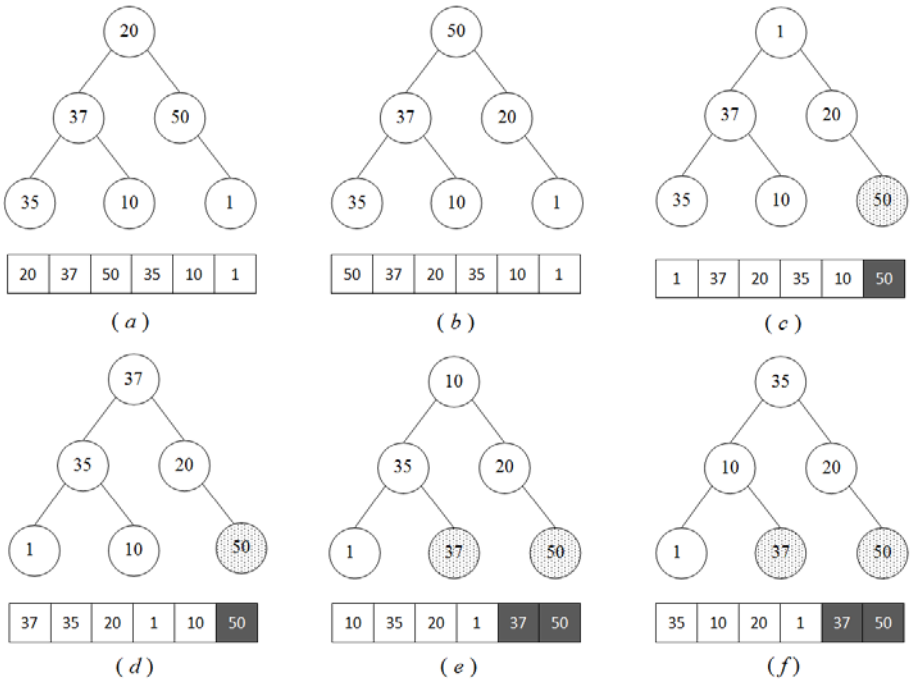


图 11-6　堆排序示例

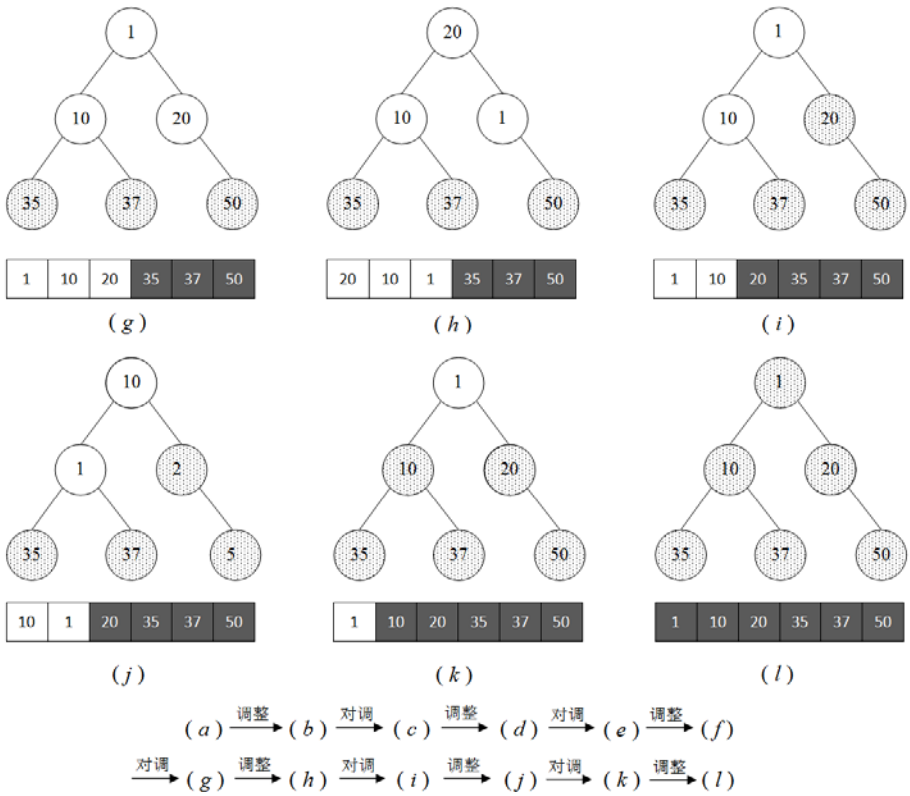


图 11-6　堆排序示例（续）

下面给出堆排序算法的示例代码。

```
//堆排序
#ifndef HEAPSORT_H
#define HEAPSORT_H

#include <vector>
using namespace std;

class HeapSort
{
private:
    int len;
    vector<int> list;
public:
    HeapSort(vector<int> _list, int _len);

    void heap_sort();
    void filterDown(int current, int last);
    void swap(int,int);
    void out();
};
#endif

#include "HeapSort.h"
#include <iostream>
using namespace std;
```

```
HeapSort::HeapSort(vector<int> _list, int _len)
{
    for (int i=0; i<_len; i++) list.push_back(_list[i]);
    this->len = _len;
}
// 堆排序
void HeapSort::heap_sort()
{
    for (int i=(len-2)/2; i>=0; i--) filterDown(i, len-1);//建立堆

    for (i=len-1; i>0; i--)
    {
        swap(0, i);
        filterDown(0, i-1);                //不断调整堆为最大堆
    }

}
// 堆的建立或调整
void HeapSort::filterDown(int current, int last)
{
    int child = 2*current+1;               // child 为 current 的子女位置
    int temp = list[current];              // 暂存子树根节点
    while (child <= last)                  // 判断是否到最后结尾
    {
        if (child<last && list[child]<list[child+1]) child++;
        //让 child 指向两子女中的大者
        if (temp >= list[child]) break;    // temp 的关键码大则不做调整
        else                               // 否则子女中的大者上移
        {
            list[current] = list[child];
            current = child;
            child = 2*child+1;                 // current 下降到子女位置
        }
    }
    list[current] = temp;                      // temp 中暂存元素放到合适位置
}

void HeapSort::swap(int i, int j)              // 交换两值
{
    int temp = list[i];
    list[i] = list[j];
    list[j] = temp;
}

void HeapSort::out()
{
    for (int i=0; i<len; i++)
    {
        cout<< list[i] << " ";
        if ((i+1)%18 == 0) cout<<endl;
    }
    cout <<endl;
}
```

可见堆排序是一树形选择排序。由于堆排序可通过树形结构保存部分比较结果，因而可以有效地减少比较次数。下面来分析一下堆排序算法的时间复杂度。

堆排序的时间主要由建立初始堆和反复重建堆这两部分的时间开销构成，它们均是通过调用堆调整算法实现的。假设堆结构对应的二叉树中有n个节点，且$2^{k-1}\leqslant n<2^k$，那么可知对应的完全二叉树有k层。在第i层上的节点数小于等于2^i，其中$i=0,1,\cdots,k-1$。在首次形成初始堆的 for 循环中，每个非叶节点都调用了一次堆调整算法，所以这个循环所用的计算时间为：

$$2\sum_{i=0}^{k-2}2^i(k-i-1)$$

其中，i是二叉树的层号，2^i是第i层的最大节点数，$(k-i-1)$是第i层节点能够移动的最大距离。对这个式子进行计算化简可以得到第一个 for 循环算法复杂度为$O(n)$。

第 2 个 for 循环中调用了$n-1$次堆调整算法，因此该循环的执行时间为$O(n\log_2 n)$。所以堆排序的时间复杂度为$O(n\log_2 n)$。堆排序的平均性能较接近于最坏性能。由于建立初始堆所需的比较次数较多，因此这种方法不适宜于对元素数较少的数据表进行排序。最后需要说明的是堆排序是一种不稳定的排序方法。

11.4 交换排序

交换排序的基本思想是两两比较待排序对象的关键码，如果发生逆序（即排列顺序与期望的相反）就交换，直到所有对象都排序完毕。本节将介绍 3 种常见的交换排序算法：冒泡排序、鸡尾酒排序和快速排序。

11.4.1 冒泡排序

算法

十大排序算法之冒泡排序。如果为应聘中最容易考到的排序算法排个序，那么它或许应该位列榜首。

冒泡排序的基本思想是模拟气泡的上浮过程。可以把数组想象成一个竖立的容器，有很多的泡泡（数据元素），大泡泡对应关键码较小的数据元素，小泡泡则对应关键码较大的数据元素，生活常识告诉我们，大泡泡会首先浮上去。

设待排序对象序列中有n个对象，首先比较待排序对象$V[n-1]$和$V[n-2]$的关键码$V[n-1].\text{key}$与$V[n-2].\text{key}$，如果$V[n-1].\text{key}<V[n-2].\text{key}$，则交换元素$V[n-1]$与$V[n-2]$，然后对$V[n-i]$和$V[n-i-1]$做同样的操作，直到对$V[0]$和$V[1]$做完操作。每进行一趟“冒泡”，结果使关键码小的对象交换到待排序对象序列的最后一个位置，其他对象也都向排序的最终位置移动。在个别情形下，对象有可能向相反方向移动，总的趋势是向排序的最终位置移动。图 11-7 演示了这个过程。

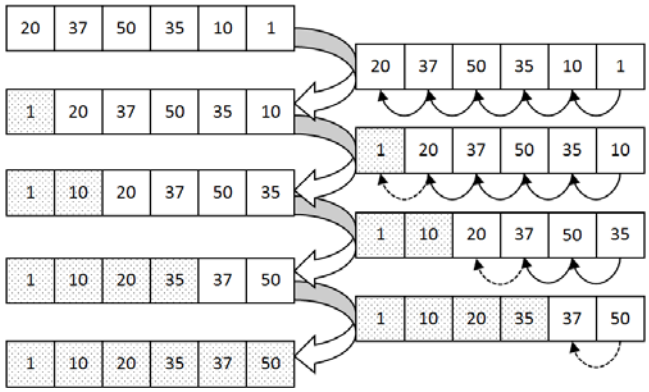


图 11-7　冒泡排序法

下面给出冒泡排序算法的示例代码。

```
// 冒泡排序
#ifndef BUBBLESORT_H
#define BUBBLESORT_H

#include <vector>
using namespace std;

class BubbleSort
{
private:
    int len;
    vector<int> list;
public:
    BubbleSort(vector<int> _list, int _len);
    void bubble_sort();
    void swap(int,int);
    void out();
};

#endif

#include "BubbleSort.h"
#include <iostream>
using namespace std;

BubbleSort::BubbleSort(vector<int> _list, int _len)
{
    for (int i=0; i<_len; i++) list.push_back(_list[i]);
    this->len = _len;
}

void BubbleSort::bubble_sort()
{
    for (int i=0; i<len; i++)                // 每次比较能使一个数归到正常位置
        for (int j=i+1; j<len; j++)          // 一次 for 循环完成一趟比较
            if (list[i] > list[j]) swap(i, j);
}

void BubbleSort::swap(int i, int j)          // 交换两个元素
{
    int temp = list[i];
```

```
    list[i] = list[j];
    list[j] = temp;
}

void BubbleSort::out() {
    //输出排序后的数组，此处代码略
}
```

可见，这样最多做$n-1$趟冒泡就能完成排序任务。但是排序特殊数组时，冒泡次数可能会少些。容易想到如果某一次冒泡没有发生任何交换，表示数组已经排好序了。因此可以在 bubble_sort()函数里增加一个标志，来标记某一次遍历是否发生了交换，如果遍历后标志为 false, 表示没有发生交换, 冒泡排序可以终止, 请读者自己尝试一下。

在最坏的情况下，关键码的比较次数为：

$$\sum_{i=1}^{n-1}(n-i)=\frac{n(n-1)}{2}$$

相应的对象移动次数为：

$$3\sum_{i=1}^{n-1}(n-i)=\frac{3}{2}n(n-1)$$

冒泡排序是一个稳定的排序方法。

算法

十大排序算法之鸡尾酒排序，名字比较另类的一种排序算法。

11.4.2 鸡尾酒排序

鸡尾酒排序又叫 Shaker 排序或双向冒泡排序，它是冒泡排序的一种轻微改进。与冒泡排序相同，Shaker 排序也是一种稳定排序算法。不同的是，普通的冒泡排序算法仅是从低到高比较序列里的每个元素，或者说普通的冒泡排序算法只能每次从前向后按一个次序进行遍历，而 Shaker 排序方法每次遍历却包括两个方向，先从前向后再从后向前，在从前往后遍历后能记录最后发生交换的两个元素位置，从后往前遍历时就从这个位置开始。这种双向交替比较不仅可以使小的浮上水面, 同时也会使大的沉到水底, 因而较普通的冒泡算法在效率上有所改进。

在 Shaker 排序法中，具有n个待排序元素的数据表将分$n/2$个阶段来完成排序操作。每个阶段的 Shaker 排序都包括一个从左向右的遍历和一个从右向左的遍历。在遍历过程中会将两个相邻元素进行比较，如果它们的顺序是非正常的就将它们做交换。

假设有待排序数组$[20,37,50,35,10,1]$，可知整个排序过程共分3个阶段。首先进行第 1 阶段的从左向右遍历，整数对(20,37)被比较，不进行交换操作。接下来再比较整数对(37,50)，它们也不进行交换。然后比较整数对$(50,35)$，由于它们处于非正常顺序，因此执行交换，得到新数组$[20,37,35,50,10,1]$。当50和10被比较时，通过一个交换得到新数组$[20,37,35,10,50,1]$。最后比较整数对(50,1)，在一个交换之后，第 1 阶段的自左向右遍历结束，并得到新数组$[20,37,35,10,1,50]$。现在进行第 1 阶段的自右向左遍历。首先比较整数对(10,1)，并做一次交换，得到新数组$[20,37,35,1,10,50]$。

然后比较整数对(35,1)，进行一次交换后得到新数组[20, 37, 1, 35, 10, 50]。如此继续下去最终在第 1 阶段的自右向左遍历结束后得到新数组[1, 20, 37, 35, 10, 50]。第 1 阶段操作完毕后，最大和最小的元素已经分别得到，排除掉它们得到数组[20, 37, 35, 10]，在这个数组上进行第 2 阶段操作。在完成自左向右遍历后，得到[20, 35, 10, 37]。接下来再进行自右向左遍历得到[10, 20, 35, 37]。同样排除掉两头的元素，因此第 3 阶段的操作将在[20, 35]上进行。图 11-8 演示了这个过程。

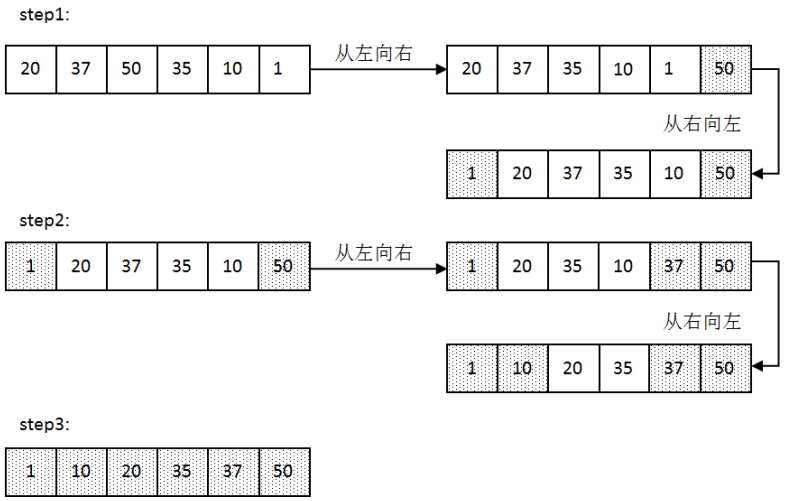


图 11-8　鸡尾酒排序

综上可以推广到一般的情况下，假设要对数据表$V[0], V[1], \cdots, V[n-1]$进行排序。那么在第 1 阶段的自左向右遍历结束后，数据表中的最大元素就位于表中的最右端的位置。因此自右向左遍历只需在新表中的第0 项至第$n-2$项中进行。在接下来的自右向左遍历结束后，表中的最小元素就位于序列中最左端的位置上了。因此下一阶段的操作只需在新表中的第1 项到第$n-2$项中进行即可。也就是说，第p个阶段的自左向右遍历只在新表中的第$p-1$到第$n-p$个位置上进行，而第p个阶段的自右向左遍历则相应地只在新表中的第$n-p-1$到第 p个位置上进行。特别地，当数据表中的元素为偶数个时，最后一阶段的自右向左遍历不进行任何操作。

实现 Shaker 排序算法的源代码如下。

```
//Shaker 排序

#ifndef SHAKERSORT_H
#define SHAKERSORT_H

#include <vector>
using namespace std;

class ShakerSort
{
private:
    int len;
    vector<int> list;
public:
```

```
    ShakerSort(vector<int> _list, int _len);
    void shaker_sort();
    void swap(int,int);
    void out();
};
#endif

#include "ShakerSort.h"
#include <iostream>
using namespace std;

ShakerSort::ShakerSort(vector<int> _list, int _len)
{
    for (int i=0; i<_len; i++) list.push_back(_list[i]);
    this->len = _len;
}

void ShakerSort::shaker_sort()
{
    int i, left = 0, right = len - 1,shift = 0;
    while(left < right)
    {
        for(i = left; i < right; i++)           // 向右进行气泡排序
        {
            if(list[i] > list[i+1])
            {
                swap(i, i+1);
                shift = i;                      // 记录最后一次所在的状态
            }
        }
        right = shift;

        for(i = right; i > left; i--)           // 向左进行气泡排序
        {
            if(list[i] < list[i-1])
            {
                swap(i ,i-1);
                shift = i;
            }
        }
        left = shift;
    }
}

void ShakerSort::swap(int i, int j)                  // 交换两个元素的位置
{
    int temp = list[i];
    list[i] = list[j];
    list[j] = temp;
}

void ShakerSort::out() {
    //输出排序后的数组，此处代码略
}
```

为了分析 Shaker 排序算法的效能，首先看看元素对之间的比较次数，来做一个分步分析。易知，在第 1 阶段，元素的比较次数为$(n-1)+(n-2)$；而在第 2 阶段，元素的比较次数为$(n-3)+(n-4)$，依此类推，最终得到所有的阶段比较次数的和为：

$$(n-1)+(n-2)+(n-3)+\cdots+i=\frac{n(n-1)}{2}$$

通过理论证明可以得到，Shaker 排序算法的复杂度是$O(n^2)$。需要说明的是，在最坏的情况和平均的情况下算法的复杂度都是$O(n^2)$。但是，如果数据表在进行排序之间已经处于正常顺序或非常接近正常的顺序，那么算法的复杂度将接近$O(n)$。

11.4.3　快速排序

算法

十大排序算法之快速排序，顾名思义，它是一种高效的算法。另外，这个算法也是本章中最不容易理解的一个算法。他的提出者霍尔也是1980年度的图灵奖得主。

快速排序也叫分区排序，它是一种平均性能非常好的排序方法。英国计算机科学家安东尼·霍尔（Charles Antony Richard Hoare）在其 1962 年发表的论文中提出了一种高效的划分交换排序算法，即著名的快速排序算法。该算法采用了一种分治的策略，其基本思想是取待排序对象序列中的某个对象为基准（比如第一个对象），按照该关键码的大小，将整个对象序列划分为左右两个子序列：左侧子序列中所有对象的关键码都小于或等于基准对象的关键码，右侧子序列中所有对象的关键码都大于基准对象的关键码，基准对象排在这两个子序列中间，然后分别对这两个子序列重复实施上述方法，直到排序完成为止。

数组分区、递归求解是快速排序算法的核心思想。如图 11-9 所示，数组的两个分区具有下面的属性，S_1分区的所有项都小于基准项p，而S_2分区的所有项都大于等于p。这个属性并不表示数组已经完成排序，但说明一个事实，即选定基准元素后，虽然从位置first到middle − 1的元素的相对位置可能会变化，但变化不会超过middleElement。同样，选择基准元素后，虽然从位置middle + 1到last的元素的相对位置可能变化，但依然在middle + 1到last的范围内。由于在最终的有序数组中，基准元素的位置保持不变，因此也可以将它称作基准项。

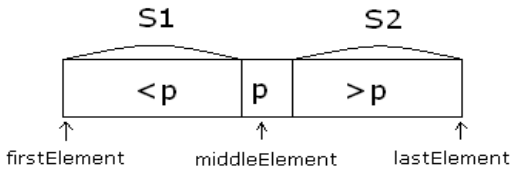


图 11-9　基准元素左右两侧的分区

数组分区引出了“ 自顶向下 ”的递归解决方案。围绕基准项排列数组，将产生两个更小的排序问题，就是对排序数组的左右两侧分别排序。具体的调用过程是在递归调用之前分解数组，在正确的位置放置基准对象。基准对象的左侧和右侧的排序问题是更小的数组排序问题，如此递归处理，直到基准对象的左侧和右侧要处理的是单个元素为止。

简单地说，快速排序算法可以被归结为如下 4 步。假设有一个待排序的数组 Array。

（1）如果数组 Array 的大小是 0 或者 1，那么返回这个数组，这就是递归的基本条件，有关基本条件的知识请参见本书第 6 章所述的内容。

（2）从数组中选择一项作为基准项。

（3）创建两个子数组。将原数组中的所有小于基准项的元素放入第 1 个子数组中；相应地，将原数组中的所有大于基准项的元素放入第 2 个子数组中。

（4）返回一个数组。这个数组由 3 部分构成，它们是应用快速排序方案排序后的第 1 个子数组，基准项以及应用快速排序方案排序后的第 2 个子数组。

由于快速排序算法使用了递归，因此容易造成理解上的困难。下面给出一个例子来说明它是如何工作的。考虑存在一个待排序数组，如图 11-10 所示。

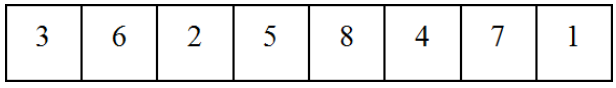


图 11-10　待排序数组

因数组中包含的元素个数大于1，因此快速排序算法进入第 2 步——选择基准项。实际应用中，存在很多不同的策略来选择基准项，其中一个比较简单的方案是选择位置上居中的元素作为基准项。由此，在上述数组中可以选择5作为基准项。在选择了基准项之后，算法将剩余的元素分配到两个子数组中。其中一个子数组的所有元素都小于5，而另一个中的元素都不小于5。图 11-11 演示了这个过程，其中基准项被着色标出。

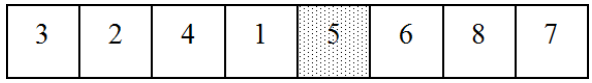


图 11-11　选择基准项并进行数组分区

从图 11-11 可知，第 1 次分区新建了两个更小的数组，然后快速排序算法会进行递归调用。注意目前两个子数组中的元素个数都大于2，因此，算法在两个子数组中选择基准项，然后把它们分成更小的数组。结果如图 11-12 所示。

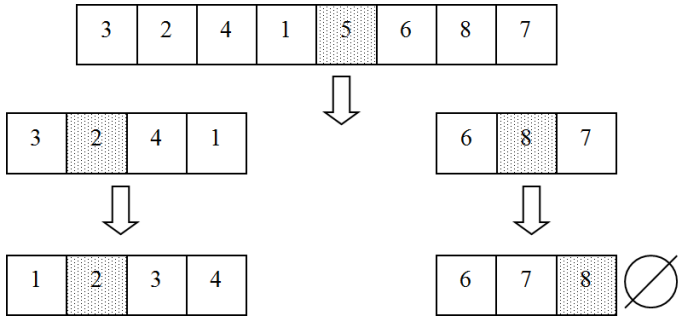


图 11-12　递归分解

快速排序算法仅需要选择基准项并对包含超过一个元素的子数组进行分区。从图 11-12 中可以看出，这里得到了 4 个子数组。第 1 个子数组只包含1个元素。第 2 个子数组包含元素3和4，第 3 个子数组包含元素6和7。第 4 个子数组不包含元素，用空集符号表示。快速排序算法不需要对第 1 个子数组和第 4 个子数组进行操作，因为它们所含的元素数目都小于2。因为第 2 个和第 3 个数组的元素数目都大于1，那么就存在它们

不处于正常顺序的可能。快速排序算法必须递归地确定基准项并进行数组分区，图 11-13 说明了这一过程。

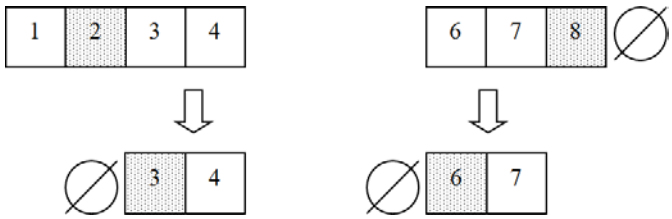


图 11-13　递归分解

快速排序算法的递归特性将原始数组分割成越来越小的子数组。如此继续下去，最终数组排序完毕。而对子数组的基准项确定和分区将最终导致整个数组的有序化。图 11-14 给出了上例的排序结果。

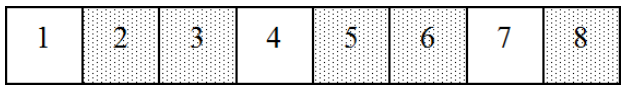


图 11-14　应用快速排序算法进行操作的结果

下面给出快速排序算法的示例代码。

```
//快速排序
#ifndef QUICKSORT_H
#define QUICKSORT_H

#include <vector>
using namespace std;

class QuickSort
{
private:
    int len;
    vector<int> list;
public:
    QuickSort(vector<int> _list, int _len);
    void quick_sort(int,int);
    void swap(int,int);
    void out();
};
#endif

#include "QuickSort.h"
#include <iostream>
using namespace std;

QuickSort::QuickSort(vector<int> _list, int _len)
{
    for (int i=0; i<_len; i++) list.push_back(_list[i]);
    this->len = _len;
}

void QuickSort::quick_sort(int left, int right)
{
```

```
    int i = left;
    int j = right;
    int pivot = list[left];

    while (i<j)
    {
        while (i<j && list[j]>=pivot) j--; //找到比基数小的元素
        if(i < j) swap(i, j);
        while (i<j && list[i]<=pivot) i++; //找到比基数大的元素
        if(i < j) swap(i, j);
    }

    if (i!=left) quick_sort(left, i-1);    //对 list[left...i-1]排序
    if (j!=right) quick_sort(j+1, right);  //对 list[j+1...right]排序
}

void QuickSort::swap(int i, int j)
{
    int temp = list[i];
    list[i] = list[j];
    list[j] = temp;
}

void QuickSort::out() {
    //输出排序后的数组，此处代码略
}
```

由于快速排序算法采用递归方式进行，因而可以构建相应的递归树。从快速排序算法的递归树可知，快速排序的趟数取决于递归树的深度。最理想的情况是每次选定基准项后，分区得到的左侧子序列和右侧子序列的长度相同，则下一步是对两个长度减半的子序列进行排序。易知，在含有n个元素的序列中，对一个对象进行定位所需时间为$O(n)$，若设$T(n)$是对n个元素序列进行排序所需时间，而且每次对一个对象正确定位后，正好把序列划分成长度相等的两个子序列，则可得此时的总计算时间为：

$$\begin{aligned} T(n) \leqslant cn + 2T\left(\frac{n}{2}\right) &\leqslant cn + 2\left[\frac{cn}{2} + 2T\left(\frac{n}{4}\right)\right] = 2cn + 4T\left(\frac{n}{4}\right) \\ &\leqslant 2cn + 4\left[\frac{cn}{4} + 2T\left(\frac{n}{8}\right)\right] = 3cn + 8T\left(\frac{n}{8}\right) \\ &\cdots\cdots \\ &\leqslant cn\log_2 n + nT(1) = O(n\log_2 n) \end{aligned}$$

通过理论证明得到快速排序算法的平均排序时间是$O(n\log_2 n)$。快速排序之所以被称为快速排序，也是因为就平均效率而言，它是所有内排序方法中最好的一个，这一点也已经得到了实验的证明。

下面来分析一下快速排序可能出现的最糟糕的情况。如果待排序序列已经按其关键码从小到大排好序，且每次选定的基准项都是子序列中最小或最大的那个元素，那么每次分区只得到一个比上一次少对象的子序列，这样必须经过$n-1$趟遍历才能把所有的元素定位完毕，而且第1趟需要经过$n-1$次关键码比较才能找到第1个对象的摆放位置，第2趟需要经过$n-2$次关键码比较才能找到第2个对象的摆放位置，依此类推，总的关键码比较次数将达到：

$$\sum_{i=1}^{n-1}(n-i)=\frac{n(n-1)}{2}\approx\frac{n^2}{2}$$

在这种情况下排序速度退化到简单排序的水平，比直接插入排序还慢。占用附加存储将达到$O(n)$。可见合理的选择基准项是非常重要的。如果每次都能够选择适当的基准项，那么就可以使每次划分所得的两个子序列中的元素个数尽可能接近，这样就可以加快排序速度，但是由于对象的初始排列次序是随机的，因此这个要求是不容易实现的。可以想到的一种改进措施是取每个待排序序列的第一个对象、最后一个对象和位置居中的对象，共计 3个对象，并取其关键码居中者作为基准对象。

最理想的情况下，快速排序所需的存储开销为$O(\log_2 n)$，最大递归调用层次数为$[\log_2(n+1)]$。快速排序的表现性能往往跟数据表中的元素个数n有关，对于n较大的平均情况而言，快速排序是“ 快速 ”的，但是当n变得很小时，快速排序方法往往比其他简单排序方法要慢。因此使用快速排序方法也要酌情而定。最后说明一点，快速排序是一种不稳定的排序方法。

11.5　归并排序

算法

十大排序算法之归并排序。

归并排序和快速排序一样，是使用分治策略实现的排序算法，算法可采用递归实现。所谓归并，就是将两个或两个以上的有序数据序列合并成一个新的有序数据序列。

首先来介绍有序数组的线性归并方法。基本的归并算法接收两个输入数组$\boldsymbol{A}$和$\boldsymbol{B}$，并输出结果数组$\boldsymbol{C}$。在这个过程中有 3 个计数器A_counter、B_counter和C_counter，它们分别对应归并过程中数组$\boldsymbol{A}$、$\boldsymbol{B}$和$\boldsymbol{C}$的下标。在算法开始时，它们都位于各自数组的始端。算法的实施过程是将$\boldsymbol{A}$[A_counter]和$\boldsymbol{B}$[B_counter]中的元素进行比较，并把较小的那个元素续填到数组$\boldsymbol{C}$中，并使计数器自增。一旦$\boldsymbol{A}$或$\boldsymbol{B}$中任意一个数组中的元素被耗尽，则把另一个数组中的剩余元素全部续填到数组$\boldsymbol{C}$中，整个归并过程就完成了。

下面举一个例子来说明这个过程。假设有数组$\boldsymbol{A}$和$\boldsymbol{B}$，如图 11-15 所示。

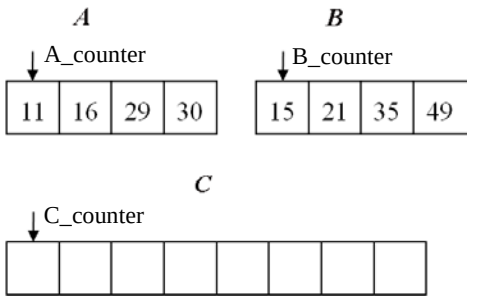


图 11-15　线性归并示例步骤 1

算法开始时，A_counter、B_counter和C_counter都等于 0，首先比较$\boldsymbol{A}$[A_counter]和$\boldsymbol{B}$[B_counter]，即比较$\boldsymbol{A}$[0]和$\boldsymbol{B}$[0]。此时发现$11<15$，则将11填入$\boldsymbol{C}$[0]处，此时将A_counter自加1。再比较$\boldsymbol{A}$[A_counter]和$\boldsymbol{B}$[B_counter]，这时$16>15$，则将15填入$\boldsymbol{C}$[1]

处，并让B_counter自加1。在以上过程中相应地，C_counter也进行自加操作，结果如图 11-16 所示。

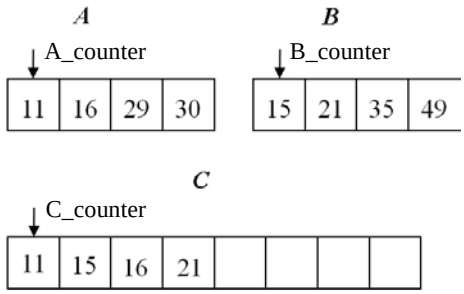


图 11-16　线性归并示例步骤 2

同理这时继续比较$\boldsymbol{A}[1]$和$\boldsymbol{B}[1]$，结果如图 11-17 所示。

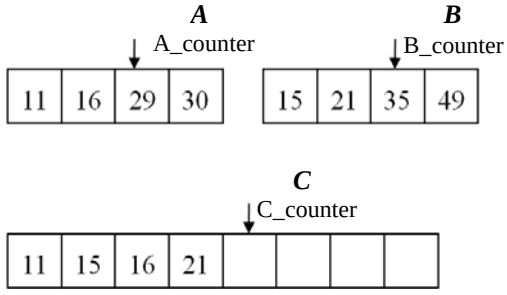


图 11-17　线性归并示例步骤 3

依据规则继续归并操作，这里不再详细叙述具体步骤，最终可以得到数组$\boldsymbol{A}$和$\boldsymbol{B}$的归并结果如图 11-18 所示。

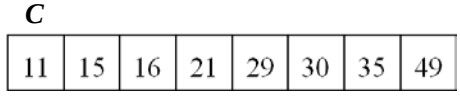


图 11-18　归并结果

从上例中可以很容易地得出归并过程的形式化描述，即归并算法将两个有序的数组合并到一个数组中并使之有序。这两个数组并不一定是相同大小的，但需要一个额外的数组存放归并结果。算法会比较两个数组中当前位置上的元素，并将较小的放入结果数组中，如此往复，如果其中一个先到达末尾，则将另外一个剩下部分放入结果数组中。而归并排序则是通过将数组不断划分（第 1 次分成两半，第 2 次分成 4 份……）的递归过程完成排序的。划分的过程直至得到仅有一个元素的数组返回为止。可以假定一个元素是有序的，那么就能够把两个数据项归并到含有两个元素的有序数组中，再次返回，将这一对两个元素的数组归并到一个 4 个元素的数组中。返回最外层的时候，这个数组将会有两个分别有序的子数组，再次归并则完成排序。形象地说就是假设数组 Array 有N个元素，那么可以看成数组Array是由N个有序的子序列组成，每个子序列的长度为1，然后再两两合并，得到了$N/2$个长度为2或1的有序子序列，再两两

合并，如此重复，直到得到一个长度为N的有序数据序列为止，利用两个有序序列的合并来实现归并排序的排序方法称为二路归并排序。

下面举例说明二路归并排序是如何工作的。假设$\{5,1,9,0,8,2,7,3,6,4\}$为待排序的初始输入序列。采用归并排序法对其进行排序。图 11-19 说明了序列在每遍扫描时的合并情况。

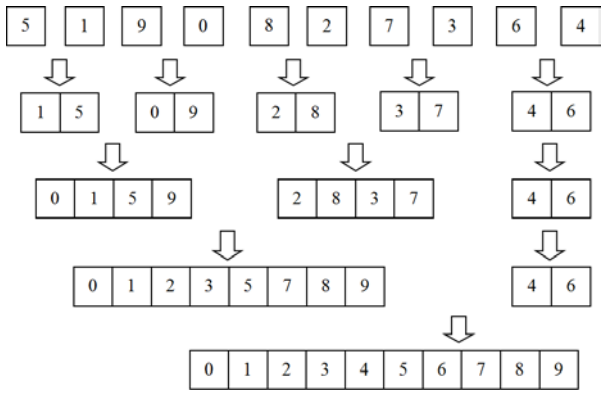


图 11-19　二路归并排序过程示例

下面给出归并排序算法的示例代码。

```
//归并排序
#ifndef MERGESORT_H
#define MERGESORT_H

#include <vector>
using namespace std;

class MergeSort
{
private:
    int len;
    vector<int> list;
    vector<int> link;
public:
    MergeSort(vector<int> _list, int _len);
    int merge_sort(int,int);
    int list_merge(int,int);
    void out();
};
#endif

#include "MergeSort.h"
#include <iostream>
using namespace std;

MergeSort::MergeSort(vector<int> _list, int _len)
{
    list.push_back(0);
    link.push_back(0);
    for (int i=0; i<_len; i++) {
        list.push_back(_list[i]);
        link.push_back(0);
```

```
    }
    this->len = _len;
}

//归并排序:递归实现
//具体方法:以 merger_link[]提供链表功能。merger_link[i]对应在有序子序列中
//merger_list[i]后一个节点在原 merger_list[]中的下标；
//merger_link[0]总是表示有序子序列开始的节点在 merge_list[]中的下标；
//st1,st2 为两个有序序列的第 1 个节点；
//将它们归并，并返回其第 1 个节点的下标，即 merger_link[0]

int MergeSort::list_merge(int st1, int st2)
{
    int k = 0, i = st1, j = st2;
    while (i && j)                      //当两序列未检测完
        if (list[i] <= list[j])         //将 merge_list[i]和 merge_list[j]
        {                               //中小的连接到 merger_link[k]后
            link[k] = i;
            k = i;
            i = link[i];
        }else
        {
            link[k] = j;
            k = j;
            j = link[j];
        }
        if (!i) link[k] = j;            //将剩余未检测完的 merge_list[]
        else link[k] = i;               //连接到 merger_link[k]后
        return link[0];
}

int MergeSort::merge_sort(int left, int right)
{
    if (left>=right) return left;
    int middle = (left + right)/2;
                                        //对左右两子序列进行归并
    return list_merge(merge_sort(left,middle), merge_sort(middle+1,right));
}

void MergeSort::out()
{
    int i = link[0];
    int j = 0;
    while (i)
    {
        j++;
        cout<< list[i] <<" ";
        i = link[i];
        if (j%18 == 0) cout <<endl;
    }
    cout <<endl;
}
```

对n个记录的序列进行归并排序时，必须进行$\lceil \log_2 n \rceil$趟归并，而每趟归并操作的时间复杂度为$O(n)$，所以二路归并排序算法时间复杂度为$O(n \log_2 n)$。与快速排序和堆排序相比，归并排序的最大优点是，它是一种稳定的排序方法，但需要说明的是在一般

情况下，很少利用二路归并排序进行内部排序。

11.6　计数排序

算法
十大排序算法之计数排序。

计数排序是一个非基于比较的线性时间排序算法。计数排序对输入的数据有附加的限制条件。

（1）输入的线性表的元素属于有限偏序集S。如果集合A上的二元关系R是自反的、反对称的和传递的，那么就称R为A上的偏序，称序偶$\langle A,R\rangle$为偏序集。

（2）设输入的线性表长度为n，$\|S\|=k$（表示集合S中元素的总数目为k），则$k=O(n)$，即k是介于线性表中数据上界与下界之间的所有值的个数。

计数排序的基本思想是，对于数据表中的每一个元素V_i，确定出小于V_i的元素个数，并据此直接确定V_i的输出位置。但是如果有多个元素具有相同的关键码时，显然不能将这些元素放在输出序列的同一个位置上。因此，需对原始方案做适当的修改。

计数排序的具体算法过程为：设输入的线性数据表V的长度为n，$V=\{V_1,V_2,\cdots,V_n\}$；数据表中的元素属于有限偏序集S，$\|S\|=k$且$k=O(n)$，$S=\{S_1,S_2,\cdots,S_k\}$；则计数排序算法首先扫描整个集合S，对每一个$S_i\in S$，在线性表V中找到等于S_i的元素的个数$T(S_i)$，并将$T(S_i)$填入临时表T中。根据临时表T建立定位表T'，建立规则是：

$$T'(i)=\begin{cases}T(i) & i=1\\ T(i)+T'(i-1) & k\geqslant i>1\end{cases}$$

然后扫描整个线性数据表V，对V中的每一个元素V_i，将V_i放在输出线性表中的第$T'(V_i)$个位置上，并将$T'(V_i)$减1。

下面举例来说明计数排序是如何工作的，假设有一个数据表V。算法首先建立临时表T并将其中各项初始化为0，输出集为空。此过程如图 11-20 所示。

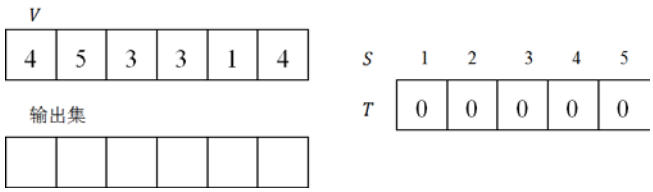


图 11-20　计数排序示例——初始化过程

扫描整个集合S，对每一个$S_i\in S$，在线性表V中找到等于S_i的元素的个数$T(S_i)$，并将$T(S_i)$填入临时表T中。这个过程如图 11-21 所示。

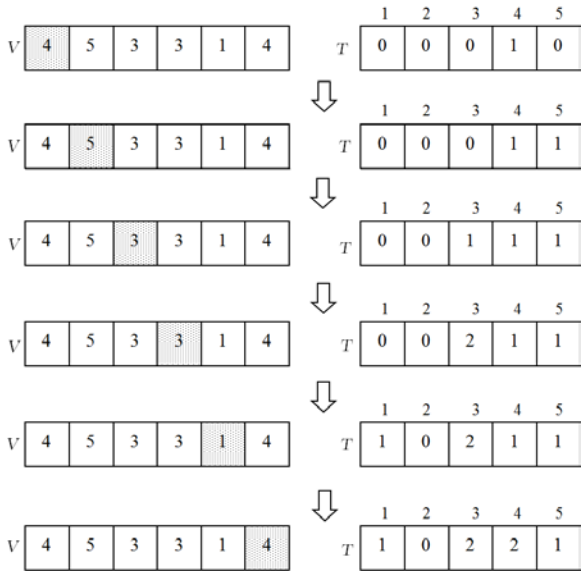


图 11-21　计数排序示例——临时表T填写过程

根据临时表T和前面给出的规则建立定位表T'，结果如图 11-22 所示。

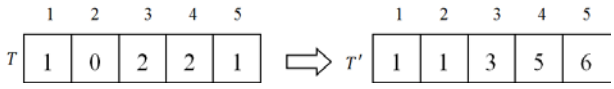


图 11-22　计数排序示例——建立定位表T'

从后向前遍历数据表V，顺序得到V_i，在定位表T'中查找对应的$T'(V_i)$，$T'(V_i)$表示的就是V_i在输出表中的位置，于是将V_i填入输出表中的$T'(V_i)$位置处，并将$T'(V_i)$自减1。这个过程如图 11-23 所示，当输入表填满时，排序工作完毕。

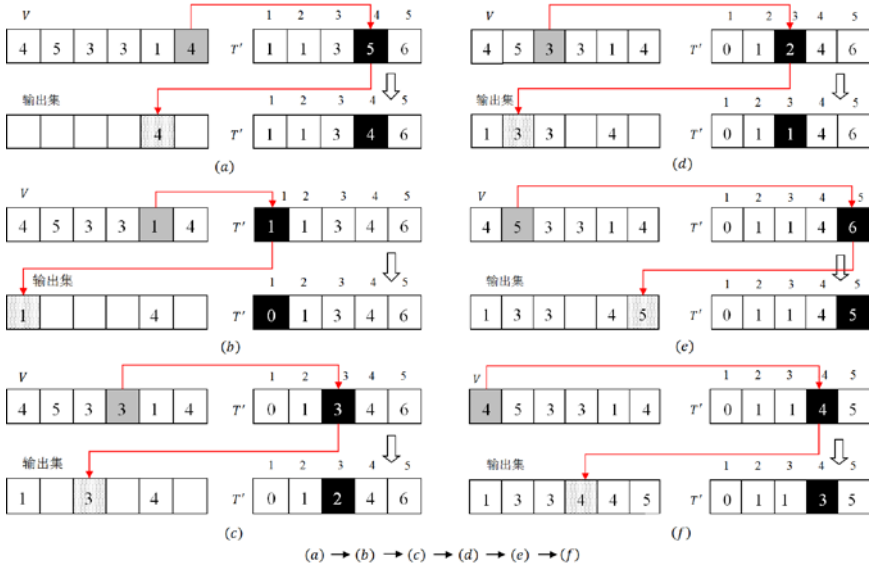


图 11-23　计数排序示例——填写输出表

下面给出计数排序算法的示例代码。

```
//计数排序
#ifndef COUNTINGSORT_H
#define COUNTINGSORT_H

#include <vector>
using namespace std;

class CountingSort
{
private:
    int len;
    int max;
    vector<int> list;
    vector<int> result;              //保存结果
public:
    CountingSort(vector<int> _list, int _len, int _max);
    void counting_sort();
    void out();
};

#endif

#include "CountingSort.h"
#include <iostream>
using namespace std;

CountingSort::CountingSort(vector<int> _list, int _len,  int _max)
{
    this->max = _max;
    this->len = _len;

    for (int i=0; i<_len; i++)
    {
        list.push_back(_list[i]);
        result.push_back(0);
        if (_list[i] > max)
        {
            cout<< "数据超出范围!"<<endl;
            return;
        }
    }
}
void CountingSort::counting_sort()
{

   int* mark = new int[max];              //标记无序表中元素
   memset(mark, 0, max * sizeof(int));

   for (int i = 0; i < len; i++) mark[list[i]]++;
    //统计元素出现次数
   for (i = 1; i < max; i++) mark[i] += mark[i-1];
    //计算元素在有序表中的位置
   for (i = len - 1; i >= 0; i--)
    {
      result[mark[list[i]] - 1] = list[i]; //通过 mark[]直接将 list[i]
      mark[list[i]]--;                     //有序存入 result[]
   }
```

```
    delete []mark;
}

void CountingSort::out() {
    //输出排序后的数组，此处代码略
}
```

计数排序的时间复杂度为$O(n)$，但这个复杂度仅在限制条件下成立。计数排序算法没有用到元素间的比较，而是利用元素的实际值来确定它们在输出数组中的位置。因此，计数排序算法不是一个基于比较的排序算法，从而它的计算时间下界也不再是$O(n\log_2 n)$。

此外，计数排序算法之所以能取得线性计算时间的上界是因为对元素的取值范围做了一定的限制，即$k = O(n)$。如果$k \neq O(n)$，就得不到线性时间的上界。显然，这种算法适合所需排列的元素取值范围不大的情况，否则会造成空间上的消耗。最后需要说明的是，计数排序是稳定的排序算法。

到此为止，我们共讨论了 10 种排序方法，它们是直接插入排序、二分插入排序、希尔排序、直接选择排序、堆排序、冒泡排序、鸡尾酒排序、快速排序、归并排序和计数排序。每种排序算法都有各自的特点和适用情况。这里给出了一个归纳表，见表 11-1，此表总结了上述 10 种排序算法的特点。

表 11-1　排序算法总结

方　法		基本思想	具体方法	时间复杂度	算法复杂情况	稳定性
插入排序	直接插入排序	每次从无序表中取出第 1 个元素，把它插入到有序表的合适位置，使有序表仍然有序	直接插入法	$O(n^2)$	简洁	稳定
	二分插入排序		二分插入	$O(n\log_2 n)$	简洁	
	希尔排序		对无序表以 gap 为步长取出所有元素，得到 gap 个子序列，并对这些子序列进行直接插入排序。然后依次递减 gap，重复上述过程，直到 gap=0 结束	不定	中	不稳定
选择排序	直接选择排序	每次从无序表中取出最小元素加入有序表后	枚举	$O(n^2)$	简洁	不稳定
	堆排序		建立最大堆或最小堆	$O(n\log_2 n)$	高	

续表

方　法		基 本 思 想	具 体 方 法	时间复杂度	算法复杂情况	稳 定 性
交换排序	冒泡排序	对无序表的元素进行两两比较，如发现两个元素是反序的，则进行交换，直到无反序的记录为止	枚举	$O(n^2)$	极简洁	稳定
	Shaker 排序		改良冒泡排序：双向进行	最坏$O(n^2)$	简洁	
快速排序			递归	平均情况 $O(n\log_2 n)$ 最坏情况 $O(n^2)$	简洁	不稳定
归并排序		将两个或两个以上有序表合并成为一个新的有序表	递归分治	$O(n\log_2 n)$	高	
计数排序		假设无序表中元素的范围为 m，对于该序列每一个元素 x，确定序列中值小于 x 的元素的个数。此方法借助散列表的思想，将元素对应到有序表中，并以此有序输出		$O(n)$	简洁	稳定

本章参考文献

[1]　朱建莉，刘宏强．常用排序算法综述．胜利油田师范专科学校学报．2002 年，第 16 卷，第 4 期.

[2]　潘金贵，顾铁成，曾俭，滕远方．现代计算机常用数据结构和算法．南京：南京大学出版社，1994.

[3]　C. A. R. Hoare. Algorithm 64: Quicksort. Communications of the ACM, Vol.4, No.7: 321, 1961.

[4]　D. L. Shell. A High-Speed Sorting Procedure. Communications of the ACM, Vol.2, No.7: 30~32, 1959.

附录

经典求职面试题目

1．分别写出 bool、int、float 和指针类型的变量 a 与“ 零 ”的比较语句。

2．用 C++写个程序，如何判断一个操作系统是 16 位的还是 32 位的？不能用 sizeof()函数。

3．101 枚硬币 100 真、1 假，真假区别在于重量。请用无砝码天平称两次给出真币重还是假币重的结论。

4．关于 const 符号常量，请说明下面 3 种描述的区别。

（1）const char *p;

（2）char const *p;

（3）char * const p;

5．什么是“ 引用 ”？申明和使用“ 引用 ”要注意哪些问题？

6．将“ 引用 ”作为函数参数有哪些特点？

7．在什么时候需要使用“ 常引用 ”？

8．将“ 引用 ”作为函数返回值类型的格式、好处和需要遵守的规则有哪些？

9．什么时候需要“ 引用 ”？

10．下面的语句正确吗？为什么？

```
int id[sizeof(unsigned long)];
```

11．下面的代码有什么问题？

```
void test1()
{
    char string[10];
    char *str1="0123456789";
    strcpy(string, str1);
}
```

12．下面的代码有什么问题？

```
void test2()
{
```

```
    char string[10], str1[10];
    for(i=0; i<10;i++)
    {
        str1[i] ='a';
    }
    strcpy(string, str1);
}
```

13. 下面代码有什么问题？

```
void test3(char* str1)
{
    char string[10];
    if(strlen(str1)<=10)
    {
        strcpy(string, str1);
    }
}
```

14. 请阐述关键字 new、delete 与 malloc、free 的联系与区别。

15. 描述内存分配方式以及它们的区别。

16. 下面的代码有什么问题？

```
void DoSomeThing(...)
{
    char* p;
    ...
    p = malloc(1024);                          //分配 1KB 的空间
    if (NULL == p)
        return;
    ...
    p = realloc(p, 2048);                      //空间不够，重新分配到 2KB
    if (NULL == p)
        return;
    ...
}
```

17. 输入一个字符串，将其逆序后输出。

18. 已知 strcpy 的函数原型是：char *strcpy(char *strDest, const char *strSrc), 其中 strDest 是目的字符串, strSrc 是源字符串。不调用 C++/C 的字符串库函数, 请编写函数 strcpy。

19. 已知 String 类定义如下：

```
class String
{
public:
    String(const char *str = NULL);            //通用构造函数
    String(const String &another);             //复制构造函数
    ~ String();                                //析构函数
    String & operater =(const String &rhs);    //赋值函数
private:
    char *m_data;                              //用于保存字符串
};
```

尝试写出类的成员函数实现。

20. 写一个在（长度为 *n* 的）字符串中寻找一个（长度为 *m* 的）子串首次出现位置的函数。

21. 一个链表的节点结构如下：

```
struct Node
{
    int data ;
    Node *next ;
};
typedef struct Node Node ;
```

（1）已知链表的头节点 head，写一个函数把这个链表逆序

（2）已知两个链表 head1 和 head2 各自有序，请把它们合并成一个链表依然有序（保留所有节点，即便大小相同）。

（3）已知两个链表 head1 和 head2 各自有序，请把它们合并成一个链表依然有序，这次要求用递归方法实现。

22. 如何判断一个单链表是否有环？(注意不能用标志位,最多只能用两个额外指针。)

23. 输入两个整数序列。其中一个序列表示栈的 push 顺序，判断另一个序列有没有可能是对应的 pop 顺序。为了简单起见，假设 push 序列的任意两个整数都是不相等的。

24. 下面的代码有什么问题？

```
class A
{
public:
    A() { p=this; }
    ~A() { if(p!=NULL) { delete p; p=NULL; } }
    A* p;
};
```

25. 用 C 写一个输入整数且倒着输出这个整数的函数，要求用递归方法实现。

26. 编写一个函数，用于返回一棵二叉树的深度。

27. 求一个有向连通图的割点，割点的定义是：如果除去此节点和与其相关的边，有向图不再连通。请描述具体算法。

28. 编写一个函数，用于判断作为输入的一个整数数组是不是某二元查找树的后序遍历的结果。如果是返回 true，否则返回 false。

29. 编写一个函数，用于判断一棵二叉排序树是否是平衡二叉树。

30. 已知一个线性表(38,25,74,63,52,48)，采用的散列函数为 $H(Key)=Key\%7$，将元素散列到表长为 7 的哈希表中存储。若采用线性探测的开放定址法解决冲突，请问在该散列表上进行等概率成功查找的平均查找长度为多少；若利用拉链法解决冲突，则在该散列表上进行等概率成功查找的平均查找长度为多少。

31. 写一个函数找出一个整数数组中第二大的数。

32. 编写一个函数，输入 *n* 个（无序的）整数，输出其中最小的 *k* 个整数。例如输入1、2、3、4、5、6、7 和 8 这 8 个整数，则最小的 4 个整数为 1、2、3 和 4。

33. 文件中有一组整数，要求排序后输出到另一个文件中。